

User-Controlled Access Management to Resources on the Web

by

Maciej Paweł Machulak

A thesis submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy



Newcastle University
Newcastle upon Tyne, UK
2014

Dedicated to my Mom, my Dad, my Sister, and most importantly, my Wife.

Acknowledgements

Some of the presented work in this thesis was a joint effort and I want to acknowledge specific people with whom I worked during my research.

The User-Managed Access (UMA) proposal, which is presented in this thesis, is researched by the User-Managed Access Work Group at Kantara Initiative which consists of academic researchers as well as industry professionals. UMA is based on the ProtectServe protocol, which was initially defined by Eve Maler, Paul Bryan and Marc Hadley, as well as other researchers from Sun Microsystems. In parallel, I developed a related protocol, called User-Managed Access Control (UMAC), which is discussed in this thesis. I did not think it was useful to continue separate work on UMAC while concurrently UMA was being developed. Therefore, this dissertation focuses on UMA and our novel UMA software design and implementation, leveraging our increased user experience, design, and understanding of issues related to development of systems similar to UMAC. The work on UMA has received numerous contributions over the years from the entire UMA Work Group, including Eve Maler (Sun Microsystems and later other companies), Paul Bryan (Sun Microsystems and later other companies), Thomas Hardjono (MIT), Domenico Catalano (Sun Microsystems and later Oracle Italy), George Fletcher (AOL), Christian Scholz (COMlounge GmbH), Jacek Szpot (Newcastle University), Łukasz Moreń (Newcastle University), Mike Schwartz (Gluu), Mark Dobrinic (Cozmanova), myself, among many others.

I have been contributing to the UMA Work Group since 2009. In particular, I have been contributing to the use cases analysis, architecture, protocol design and several implementations. I have also held various leadership positions in this work group. Most importantly, since May 2010 I have been the vice-chair, use cases editor, and implementation coordinator of this work group. I have been engaged in advocating UMA to academic, developer, and identity communities.

Furthermore, certain parts of the software implementations, which are demonstrated in this thesis, have been done in collaboration with my colleagues who worked with me on the JISC/HEFCE funded SMART (Student-Managed Access to Online Resources) project at Newcastle University. I was leading the research on this project as well as managing the research group at the University. Firstly, Łukasz Moreń mostly worked with me on design and implementation of SMARTAM V1/V2, UMA/j, OAuth Leeloo, and other software; Jacek Szpot was mostly responsible for implementation of Puma that was based on our design and experience with UMA/j and he was also responsible for implementing Python demo applications that are discussed in this thesis.

The user study of the implemented Authorisation Manager was a joint effort with MSc student Iain Carter (Newcastle University) and other researchers of the SMART project where

I was responsible for the design of this study. Also, I acknowledge contributions from Maciej Wolniak (Newcastle University) and Domenico Catalano who worked with me on improving the User Experience of the implemented Authorisation Manager. Maciej Wolniak was also responsible for implementing SMARTAM V2 user interfaces based on our research and findings and he implemented user interfaces of some the presented demo applications.

The work on the Street Identity proposal, which is mentioned in this thesis, was done during my two placements at Google. Importantly, the initial proposal was first defined by the identity team at Google, including Eric Sachs, Andrew Nash, Breno de Medeiros, and Marius Scurtescu, among others.

Furthermore, I would like to thank numerous people who influenced my research and allowed my PhD studies to be an amazing experience.

Most importantly, I would like to thank my supervisor, Professor Aad van Moorsel, who was most helpful during my research on access control solutions. Aad shared his valuable knowledge and his time to provide me with the necessary guidance on my scientific research. He always motivated me to never settle and always push my work forward and he provided me with a learning environment where I could develop personally and professionally. Without a doubt, throughout the years of my project, he was not only a mentor but became a friend as well.

Further thanks go to my colleagues from the User-Managed Access Work Group. Firstly, I wish to thank Eve Maler who always found time for "yet another discussion" on improving the UMA proposal and taught me to think about "a bigger picture". Secondly, I would like to thank Paul Bryan for introducing me to the emerging ProtectServe (renamed to UMA) proposal in the first place. I would also like to thank Thomas Hardjono with whom I had numerous discussions on the protocol specification and I am grateful for support from Domenico Catalano who visited my research group in Newcastle and worked with us on our software implementations.

I would like to specially thank Łukasz Moreń who joined me during the SMART project and provided valuable input to its software engineering side. It was his passion for building great and robust software that contributed to the success of our work.

Special thanks go to Marius Scurtescu who was my mentor at Google and spent his time on helping me better understand digital identity on large scale. I am very grateful for numerous conversations when we discussed dozens of different identity protocols. Further thanks go to Breno de Medeiros who was an exceptional team leader at Google and taught me to balance my research between functionality, security and usability. I wish not to forget support from other colleagues at Google, including Eric Sachs, Andrew Nash, Naveen Agarwal, David Huska, David Primmer, Chunlei Niu, and others.

I am forever indebted to my family for their encouragement and support when it was most

required. In particular, I wish to thank my parents, Krystyna and Eugeniusz, for giving me the necessary strength to start my PhD journey in the first place. I am also very grateful to my sister, Anna, who taught me to be strong and always work towards my dreams.

And last but not least, I wish to thank my beloved wife Marzena for her endless support, her help, and love during my research journey. She always motivated me and encouraged me to achieve goals even when achieving them seemed impossible. I am very grateful that I can share my love for science with her.

Abstract

The rapidly developing Web environment provides users with a wide set of rich services as varied and complex as desktop applications. Those services are collectively referred to as "Web 2.0", with such examples as Facebook, Google Apps, Salesforce, or Wordpress, among many others. These applications are used for creating, managing, and sharing online data between users and services on the Web. With the shift from desktop computers to the Web, users create and store more of their data online and not on the hard drives of their computers. This data includes personal information, documents, photos, as well as other resources. Irrespective of the environment, either desktop or the Web, it is the user who creates the data, who disseminates it and who shares this data. On the Web, however, sharing resources poses new security and usability challenges which were not present in traditional computing. Access control, also known as authorisation, that aims to protect such sharing, is currently poorly addressed in this environment. Existing access control is often not well suited to the increasing amount of highly distributed Web data and does not give users the required flexibility in managing their data.

This thesis discusses new solutions to access control for the Web. Firstly, it shows a proposal named User-Managed Access Control (UMAC) and presents its architecture and protocol. This thesis then focuses on the User-Managed Access (UMA) solution that is researched by the User-Managed Access Work Group at Kantara Initiative. The UMA approach allows the user to play a pivotal role in assigning access rights to their resources which may be spread across multiple cloud-based Web applications. Unlike existing authorisation systems, it relies on a user's centrally located security requirements for these resources. The security requirements are expressed in the form of access control policies and are stored and evaluated in a specialised component called Authorisation Manager. Users are provided with a consistent User Experience for managing access control for their distributed online data and are provided with a holistic view of the security applied to this data. Furthermore, this thesis presents the software that implements the UMA proposal. In particular, this thesis shows frameworks that allow Web applications to delegate their access control function to an Authorisation Manager. It also presents design and implementation of an Authorisation Manager and discusses its evaluation conducted with a user study. It then discusses design and implementation of a second, improved Authorisation Manager. Furthermore, this thesis presents the applicability of the UMA approach and the implemented software to real-world scenarios.

Publications

Workshops and Conferences

- M. P. Machulak, Ł. Moreń, and A. van Moorsel. Design and Implementation of User-managed Access Framework for Web 2.0 Applications. In Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing, pages 1–6. ACM, 2010.
- M. P. Machulak, E. L. Maler, D. Catalano, and A. van Moorsel. User-Managed Access to Web Resources. In Proceedings of the 6th ACM workshop on Digital identity management, pages 35–44. ACM, 2010.
- M. P. Machulak and A. van Moorsel. Architecture and Protocol for User-Controlled Access Management in Web 2.0 Applications. Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems Workshops, pages 62–71, 2010.
- M. P. Machulak, E. Maler, D. Catalano, and A. van Moorsel. User-Managed Access to Web Resources. In IEEE SSP’10: Proceedings of the 2010 IEEE Symposium on Security and Privacy, Oakland, California, USA, 2010. (Poster Abstract)
- M. P. Machulak and A. van Moorsel. A Novel Approach to Access Control for the Web. IEEE SSP’09: Proceedings of the 2009 IEEE Symposium on Security and Privacy, 2009. (Poster Abstract)
- M. P. Machulak, S. Parkin, and A. van Moorsel. Architecting Dependable Access Control Systems for Multi-domain Computing Environments. Architecting Dependable Systems VI, pages 49–75, 2009.
- M. P. Machulak, J. J. Halliday, and M. C. Little. Metadata Support for Transactional Web Services. In EDOC Conference Workshop, 2007. EDOC’07. Eleventh International IEEE, pages 53–56. IEEE, 2007.

Technical Reports

- M. P. Machulak, Ł. Moreń, and A. van Moorsel. Design and Implementation of User-managed Access Framework for Web 2.0 Applications. Technical Report CS-TR-1221, Newcastle University, October 2010.

- M. P. Machulak, J. J. Halliday, and M. C. Little. Metadata Support for Transactional Web Services. Technical Report CS-TR-1158, Newcastle University, August 2010.
- M. P. Machulak, E. L. Maler, D. Catalano, and A. van Moorsel. User- Managed Access to Web Resources. Technical Report CS-TR-1196, Newcastle University, March 2010.
- M. P. Machulak and A. van Moorsel. Use Cases for User-Centric Access Control for the Web. Technical Report CS-TR-1165, Newcastle University, August 2009.
- M. P. Machulak and A. van Moorsel. A Novel Approach to Access Control for the Web. Technical Report CS-TR-1157, Newcastle University, July 2009.
- M. P. Machulak and A. van Moorsel. Architecting Dependable Access Control Systems for Multi-Domain Computing Environments. Technical Report CS- TR-1156, Newcastle University, July 2009.

Abbreviations

HAT / PAT - Host Access Token / Protection API Token

RAT / AAT - Requester Access Token / Authorisation API Token

RPT - Requester Permission Token

AT - Access Token

RT - Refresh Token

RP - Relying Party

AP - Attribute Provider

UMAC - User-Managed Access Control

UMA - User-Managed Access

AM - Authorisation Manager

API - Application Programming Interface

UX - User Experience

UI - User Interface

IDP - Identity Provider

REST - Representational State Transfer

OIDC - OpenID Connect

PEP - Policy Enforcement Point

PIP - Policy Information Point

PAP - Policy Administration Point

PDP - Policy Decision Point

STS - Security Token Service

Contents

Acknowledgements	i
Abstract	iv
Publications	v
Abbreviations	vii
List of Figures	xvii
List of Tables	xxii
1 Introduction	1
1.1 Motivation	4
1.1.1 Scenario	5
1.1.2 Shortcomings	7
1.1.3 Requirements	11
1.2 Goal	12
1.3 Approach	13
1.4 Novelty of Approach	14
1.5 Contributions	16
1.6 Thesis Outline	19
2 Background	23
2.1 Introduction	23
2.2 Distributed Environments	25
2.2.1 Multi-Domain Computing Environments	26
2.2.2 Web Environment	28
2.3 Access Control in Distributed Environments	29

2.3.1	Policies	30
2.4	Delegated Authorisation	32
2.4.1	Interactions	34
2.5	Access Control Mechanisms	36
2.5.1	Capability-Issuing Security Architecture	37
2.5.2	Decision-Issuing Security Architecture	39
2.6	Access Control Challenges	40
2.6.1	Policy Challenges	41
2.6.1.1	Heterogeneity and Distribution of Subjects and Objects	42
2.6.1.2	Context and Content-Based Access to Resources	44
2.6.1.3	Policy Heterogeneity Management	45
2.6.1.4	Policy Conflict Resolution	46
2.6.2	Architectural Challenges	48
2.6.2.1	Interoperability Between Access Control Components	48
2.6.2.2	Location of Policy Decision Points	50
2.6.2.3	Management of Access Control Systems	50
2.6.2.4	Communication Performance	52
2.6.2.5	Autonomy of Administration Domains	54
2.6.2.6	Access Control Delegation	54
2.6.2.7	Attribute Aggregation	55
2.6.2.8	Authorisation Credential Issuing and Validation	56
2.6.2.9	Assignment of Arbitrary Authorisation Attributes to Users	57
2.6.2.10	Security of Access Control Systems	58
2.7	Chapter Summary	59
3	Related Technologies	60
3.1	Introduction	60
3.2	XACML	61
3.2.1	XACML in Multi-domain Computing Environments	63
3.3	SAML	64
3.4	Kerberos	65
3.5	OAuth 1.0	66
3.6	OAuth 2.0	68
3.7	OpenID Connect	71
3.8	Street Identity	72

3.8.1	Architecture	72
3.8.2	Protocol	75
3.8.3	Discussion	79
3.9	Sticky Policies	80
3.10	VOMS	81
3.11	Locker Project	81
3.12	Menagerie	82
3.13	Secure Web 2.0 Sharing Beyond Walled Gardens	84
3.14	Secure File System for the Web	86
3.15	Fine-grained Access Control Policies for Social Networking Applications	88
3.16	Chapter Summary	88
4	User-Managed Access Control	89
4.1	Introduction	89
4.2	Initial Proposal	91
4.2.1	Architecture	92
4.2.2	Interactions	92
4.2.3	Discussion	93
4.3	Refined Proposal	95
4.3.1	Architecture	96
4.3.2	Protocol	99
4.3.3	Implementation	105
4.3.4	Advantages	107
4.3.5	Extensions	107
4.3.6	Limitations	109
4.4	Chapter Summary	110
5	User-Managed Access	111
5.1	Introduction	111
5.2	Requirements Analysis	113
5.3	Architecture	114
5.4	Delegation Protocol	117
5.4.1	User introduces host to AM	118
5.4.1.1	Host learns the location of the user's preferred AM	119
5.4.1.2	Host discovers AM's functionality	121
5.4.1.3	Host registers dynamically at AM	122

5.4.1.4	User authorises the Host for their AM	122
5.4.1.4.1	Host obtains initial authorisation in form of a code. . .	122
5.4.1.4.2	Host exchanges the code for an access token for AM. . .	124
5.4.2	User registers resources for protection at AM	125
5.4.3	User defines access control policies at AM	131
5.4.4	Requester gets access token from AM	132
5.4.4.1	Access request did not contain an RPT	133
5.4.4.1.1	Requester authorisation at AM.	133
5.4.4.1.2	Requester obtains a new RPT for a Host at AM.	135
5.4.4.2	Access request contained an invalid RPT	136
5.4.5	Requester wields access token at Host to gain access	139
5.5	Trust Model	140
5.5.1	Subject Registration	141
5.5.2	Host Registration at AM	143
5.5.3	Trusted Claims Gathering	144
5.5.4	Trust Delegation and Trust Chain	145
5.6	Credentials	146
5.7	Claims	146
5.7.1	Claims 2.0	148
5.7.2	Trusted Claims	148
5.7.2.1	OpenID Connect Claims	149
5.8	Protected Information	150
5.9	Evaluation	154
5.10	Limitations	155
5.11	Chapter Summary	159
6	Host and Requester Frameworks	160
6.1	Introduction	160
6.2	UMA/j	163
6.2.1	UMA/j Common	164
6.2.2	Discovery	165
6.2.3	Dynamic Registration	165
6.2.4	OAuth 2.0	166
6.2.5	UMA/j Host	167
6.2.5.1	Resource Registration	169

6.2.5.2	Request Filter	171
6.2.5.3	Token Validation	172
6.2.5.4	Core	173
6.2.6	Requester	173
6.2.6.1	Requester Filter	174
6.2.6.2	Token Retrieval	175
6.2.6.3	Core	177
6.2.7	Limitations	177
6.3	Puma	178
6.3.1	Framework Structure	179
6.3.1.1	Puma.UMA.*	179
6.3.1.2	Puma.OAuth.*	179
6.3.1.3	Puma.Util.*	180
6.3.1.4	Puma.Storage.*	180
6.3.1.5	Puma.Pouches.*	180
6.3.2	Common	181
6.3.2.1	Discovery	181
6.3.2.2	Dynamic Registration	181
6.3.2.3	OAuth 2.0-based Authorisation	182
6.3.3	Host	183
6.3.3.1	Data Model	183
6.3.3.2	Resource registration	184
6.3.3.2.1	Resource Registration Handler.	185
6.3.3.2.2	Share button iframe.	186
6.3.3.3	Token status validation	187
6.3.3.3.1	The Warden.	188
6.3.4	Requester	189
6.3.4.1	Data Model	190
6.3.4.2	Protected Resource Access	192
6.3.4.3	Obtaining Requester Access Token	192
6.3.4.4	Obtaining Requester Permission Token	193
6.3.5	Limitations	196
6.4	Chapter Summary	197

7	Authorisation Managers	198
7.1	Introduction	198
7.2	SMART Authorisation Manager V1	200
7.2.1	Architecture	201
7.2.2	Security	202
7.2.3	Support for User-Managed Access	203
7.2.3.1	Authorisation tokens	203
7.2.3.2	Support for host applications	204
7.2.3.3	Support for requester applications	204
7.2.4	Policy Model	206
7.2.5	Policy Evaluation	207
7.2.5.1	Claims	208
7.2.5.2	Authorisations	209
7.2.6	User Interface	211
7.2.7	Integration with Applications	215
7.2.8	Implementation	220
7.2.9	Limitations	220
7.3	Evaluation of SMARTAM V1 User Interface	223
7.3.1	Research	223
7.3.1.1	Usability Factors	224
7.3.1.2	Research Method	224
7.3.1.3	User Scenario	226
7.3.1.4	Data Collection	227
7.3.2	Research Results	227
7.3.2.1	Feedback from the respondents	228
7.3.2.2	Interpretation of the results	229
7.3.2.3	Interface inconsistencies	230
7.3.2.4	UMA-protocol understanding	231
7.3.3	Shortcomings and Recommendations	231
7.3.3.1	Analysis of usability factors	232
7.3.3.2	Shortcomings	233
7.3.3.3	Recommendations	234
7.4	SMART Authorisation Manager V2	236
7.4.1	Architecture	237
7.4.1.1	Persistence and Services Layer	237

7.4.1.2	API Layer	238
7.4.1.3	Presentation Layer	239
7.4.2	Security	240
7.4.3	Support for User-Managed Access	240
7.4.3.1	Authorisation Tokens	241
7.4.3.2	Support for Host Applications	242
7.4.3.3	Resource Registration	242
7.4.3.4	Support for Requester Applications	243
7.4.4	Federated Authentication	245
7.4.5	Policy Model	246
7.4.5.1	Database Implementation of Policies	248
7.4.6	Policy Evaluation	250
7.4.6.1	Issuing empty RPTs to Requesters	250
7.4.6.2	Registering Permissions	250
7.4.6.3	Obtaining Authorisation	251
7.4.6.3.1	Authorising User acting as Requesting Party.	252
7.4.6.3.2	Requesting Party added manually by Authorising User.	253
7.4.6.3.3	Any Requesting Party.	254
7.4.6.4	Validating Tokens from Requesters	254
7.4.7	Access History	255
7.4.8	User Interface Description	256
7.4.8.1	Scenario	256
7.4.8.1.1	Login at Host and AM.	257
7.4.8.1.2	Register a document for protection at AM using host application.	257
7.4.8.1.3	Choose contacts to share the document with.	259
7.4.8.1.4	Choose applications to be used by contacts for accessing the document.	260
7.4.8.1.5	Setup permissions for contacts and applications.	260
7.4.8.1.6	Share the document.	262
7.4.8.2	Main User Interface	263
7.4.8.3	Comparison	271
7.4.9	Implementation Details	275
7.4.9.1	Presentation Layer	276
7.4.9.2	Services Layer	277

7.4.9.3	Persistence Layer	277
7.4.9.4	Application Programming Interface	277
7.4.9.5	Security	278
7.4.10	Limitations	278
7.5	Chapter Summary	281
8	Conclusions	282
8.1	Thesis Summary	282
8.2	Future Work	284
8.2.1	Deployment and acceptance of user managed access control approaches .	285
8.2.2	Development of improvements to UMA protocol	286
8.2.2.1	Optimised integration with client applications	286
8.2.2.2	Schema for protected resources	286
8.2.2.3	Protected resources discovery support	287
8.2.2.4	User's preferred AM discovery support	287
8.2.3	Development of improvements to SMARTAM implementation	288
8.2.3.1	Extended audit with support for arbitrary events	288
8.2.3.2	Extended policy support	288
8.2.3.3	Visualisation of complex policies	289
8.2.3.4	User studies of the newly implemented AM	289
8.2.3.5	Integration of AM with federated authentication proposals . . .	290
8.2.3.6	Mobile applications support	291
8.3	Concluding Remarks	291
	Appendices	292
	Appendix A Graphical representation of UMA/j module dependencies	293
	Appendix B Example UMA/j Host configuration file	296
	Appendix C UMA/j Implementation and Evaluation	299
C.0.1	Implementation	299
C.0.2	Evaluation	299
	Appendix D Puma Implementation and Evaluation	302
D.0.3	Implementation	302
D.0.4	Evaluation	302
D.0.4.1	Host application	303

D.0.4.2 Requester application	306
Appendix E SMARTAM V1 Data Model	308
Appendix F SMARTAM V2 Data Model	310
Appendix G Example UMA Authorisation Manager Configuration Data	312
Appendix H Questionnaire for SMARTAM V1 UI Research Study	313
Appendix I Sharing Trustworthy Personal Data with Future Employers Scenario	315
Appendix J Mind map of access control for the open Web	324
Bibliography	327

List of Figures

1.1	Scenario highlighting interactions and the flow of data between Web applications.	6
1.2	Distributed access control policies created and managed in example scenario. . .	10
1.3	World map showing visitor demographics of the OAuth Leeloo website at [117] between 1st August 2010 and 29th April 2013. Map has been generated using the Google Analytics software [18].	18
1.4	Visualisation of the thesis outline.	20
2.1	Overlap of access control requirements in different deployment environments. . .	25
2.2	Multi-domain computing environment forming a Virtual Organisation [219]. . . .	27
2.3	The Internet & the Web environment.	28
2.4	Agent model approach to access control [219].	36
2.5	Capability-issuing approach to authorisation in computing environments [219]. .	38
2.6	Decision-issuing approach to authorisation in computing environments [219]. . .	39
2.7	A high-level view of the Policy Administration Point / Policy Syndication Server hierarchy.	53
3.1	XACML data-flow diagram [97].	62
3.2	Interactions between parties in the Kerberos protocol [273].	66
3.3	OAuth 1.0 protocol flow [200].	67
3.4	OAuth 2.0 - Authorisation Code Grant (former Web Server profile) [202].	70
3.5	The Google's Street Identity proposal.	73
3.6	Access tokens involved in the Street Identity protocol.	75
3.7	Example of data organisation in PC-centric vs. Web-centric world [195].	83
3.8	The system architecture of the proposed Web 2.0 content sharing solution [286].	85
4.1	Research path on proposals for user-managed access control solutions for the Web.	90
4.2	High level view of the user-centric access control system for the Web.	91

4.3	Basic Architecture for User-Managed Access Control for the Web [244].	96
4.4	Comparison of relations between parties of a user-managed access control solution with XACML [245].	99
4.5	High-level overview of the UMAC protocol [244].	100
4.6	UMAC Step 1: A User establishes a trust relationship between a Host and AM. .	101
4.7	UMAC Step 3: A Requester obtains authorisation token from AM.	103
4.8	UMAC Step 4: A Requester attempts access with authorisation token.	104
5.1	Interactions between entities involved in the UMA protocol [238].	115
5.2	Sharing use cases in User-Managed Access [178].	118
5.3	High-level overview of the User-Managed Access protocol [238].	119
5.4	UMA Step 1: User introduces Host to AM.	120
5.5	User-Managed Access API Endpoints available on the Authorisation Manager. .	121
5.6	OAuth 2.0 authorisation page displayed to the User.	123
5.7	UMA Step 2: Host registers resources for protection at AM and directs a User to the policy URI at AM.	126
5.8	Access requests issued by Requesters can be subject to different protocol paths, depending on the presence and validity of the RPT.	132
5.9	UMA Step 3: Requester gets RAT/AAT and RPT tokens from AM.	134
5.10	UMA Step 4: Requester wields RPT at Host to gain access to a protected resource.	137
5.11	User-Managed Access Trust Model [157].	141
5.12	Bootstrapping Trust in User-Managed Access [156].	142
5.13	Trust Delegation and Trust Chain in User-Managed Access [157].	145
5.14	UMA conceptual model with Trusted Claims [155].	150
6.1	High-level overview of the modules in the UMA/j framework and related libraries.	163
6.2	Design of the UMA/j framework and its integration with a Web application [239].	168
6.3	Structure of the Puma framework [123; 124].	180
6.4	High-level architecture of Puma integration with a host application [123]. . . .	183
6.5	Simplified ERD diagram of using Puma framework in a host application [123]. .	184
6.6	Flow using Resource Registration Handler [123].	185
6.7	Message flow through the Puma Warden [123].	189
6.8	Simplified ERD diagram of using Puma framework in a requester application [124].	191
7.1	Research path on Authorisation Manager implementations.	198
7.2	Architecture of SMART Authorisation Manager V1.	201

7.3	Example implementation of an access control policy.	206
7.4	Example of a resource linked with an access control policy.	207
7.5	Main page of the SMARTAM V1 UI.	211
7.6	View displaying registered applications at AM.	212
7.7	View displaying registered resources at AM.	213
7.8	View displaying a list of Requesting Parties created by an Authorising User. . . .	213
7.9	View displaying list of available restrictions (claims).	214
7.10	View for creating new access control policies.	215
7.11	View for creating new rules to be added to an access control policy.	216
7.12	View for associating existing restrictions (claims) with an access control policy (drag and drop).	216
7.13	View for associating an access control policy with a resource (drag and drop). . .	217
7.14	User Interface of the requester application that allows the Requesting Party to provide their credentials. These credentials are used for authentication at AM. .	219
7.15	User Interface of the requester application that allows the Requesting Party to provide self-asserted claims. These claims are sent to AM for evaluation.	219
7.16	User scenario used during research study of the SMARTAM V1 UI.	226
7.17	Results regarding the general ease of use of the SMARTAM V1.	228
7.18	Visualisation of shortcomings of the SMARTAM V1 UI.	234
7.19	Architecture of SMART Authorisation Manager V2.	237
7.20	Overview of the User Interface architecture of SMARTAM V2.	239
7.21	SMARTAM V2 integrated with Facebook.	245
7.22	Entity Relationship Diagram of access control policies in SMARTAM V2.	248
7.23	Authorisation page that allows the Requesting Party to authorise a requester application to access a protected resource stored on a Host.	253
7.24	Request for access page that allows the Requesting Party to request access to a protected resource stored on a Host.	254
7.25	User scenario used for discussion of the SMARTAM V2 User Interface.	257
7.26	Resource and a Share button rendered for this resource on a Host.	258
7.27	View presenting myself-type policy provided by SMARTAM V2 and accessed di- rectly from a host application.	258
7.28	View presenting others-type policy provided by SMARTAM V2 and accessed di- rectly from a host application.	259
7.29	Access control policy view with missing contacts.	260
7.30	View for selecting contacts to be used in a policy.	261

7.31	View for selecting applications and actions for these applications to be used in a policy.	261
7.32	Example email that could be sent to a Requesting Party notifying them about a newly shared resource.	262
7.33	Information sent to the Facebook wall of a Requesting Party notifying them about a newly shared resource.	262
7.34	Main view of SMARTAM V2 presenting resources registered for protection. . . .	264
7.35	View presenting the resource registered for protection on SMARTAM V2.	264
7.36	View presenting a specified myself type access control policy.	265
7.37	View presenting a specified others type access control policy.	265
7.38	View presenting a specified others type access control policy with application details.	266
7.39	View presenting a specified others type access control policy with application and permission details.	267
7.40	View showing the list of contacts.	268
7.41	View presenting resources shared with a Requesting Party.	268
7.42	View showing the form that allows the user to add new contacts manually.	269
7.43	View showing the list of registered applications at AM.	269
7.44	View showing resources shared with a particular application.	269
7.45	View showing access history list (audit log).	270
7.46	View showing requests to access protected data.	270
7.47	Request for access mechanism implemented by SMARTAM V2.	271
7.48	Visualisation of improvements to the User Interface introduced in SMARTAM V2.	272
7.49	Prototype view of UI for SMARTAM V2 allowing the user to specify access control policy in a single screen.	274
8.1	Visualisation of UMA Connection concept [155].	290
8.2	User Interfaces of a prototype mobile application integrated with SMARTAM V2.	291
A.1	UMA/j Host module dependency graph.	293
A.2	UMA/j Requester module dependency graph.	294
A.3	UMA/j Authorisation Manager module dependency graph.	295
C.1	User Interface of a Web application accessing UMA-protected resource.	301
D.1	Python applications integrated with SMARTAM V2 using the Puma framework.	303
D.2	UMA-protected RESTful Web API of the PDS application.	304

E.1	SMART Authorisation Manager V1 Entity Relationship Diagram.	309
F.1	SMART Authorisation Manager V2 Entity Relationship Diagram.	311
I.1	Step 1 - Sean attempts to access his online data at the S3P application. Sean visits the S3P Web application at Newcastle University. Selects “Newcastle University”.	316
I.2	Step 2 - Sean signs in to Newcastle University with his existing credentials.	317
I.3	Step 3 - Sean sees his personal data stored by the S3P application.	317
I.4	Step 4 - Sean can view his academic record. Sean then attempts to apply for a job position at the CareerMonster Web application.	318
I.5	Step 5 - Sean visits the CareerMonster Web application. Selects to sign in with his University’s account.	318
I.6	Step 6 - Sean signs in to Newcastle University with his existing credentials.	319
I.7	Step 7 - Sean is presented with a list of available job positions. He attempts to apply for the “Junior Software Engineer” position.	319
I.8	Step 8 - In order to apply for a job position, Sean has to fill in some information or import it from S3P.	320
I.9	Step 9 - Sean is presented with a consent page, where he is required to authorize CareerMonster to access his personal data from S3P. This consent is represented on SMARTAM in form of sharing settings for Sean’s name.	320
I.10	Step 10 - Sean’s personal data is retrieved by CareerMonster directly from S3P. Sean then selects to import his academic record from S3P as well.	321
I.11	Step 11 - Sean is presented with a consent page, where he is required to authorize CareerMonster to access his personal data from S3P. This consent is represented on SMART in form of additional sharing settings (this time for Sean’s academic record).	321
I.12	Step 12 - Sean’s academic record is retrieved by CareerMonster directly from S3P. At this point, all required information has been submitted to CareerMonster and Sean can apply for the job position of his choice.	322
I.13	Step 13 - Sean can sign in to SMART and view various information regarding his University’s data and how this data is shared with other applications.	322
I.14	Step 14 - Sean can view sharing settings for his academic record.	323
I.15	Step 15 - Sean can view how his data is shared with other applications.	323
J.1	Mind map of access control on the open Web - part 1.	325
J.2	Mind map of access control on the open Web - part 2.	326

List of Tables

2.1	Relationships between access control challenges and requirements for a user-managed access control proposal.	42
5.1	Operations supported by Authorisation Manager Resource Registration endpoint.	128
5.2	Parameters of a resource set description.	129
5.3	Parameters of an action description.	130
5.4	Relationships between tokens and actors of the UMA proposal.	147
6.1	Comparison of UMA/j and Puma frameworks.	161
7.1	UMA-enabled Web applications developed during the course of research and their integration with developed Authorisation Managers.	200

List of listings

1	Example of a SAML Authorisation Decision Statement.	65
2	Example of HTTP request, carrying token credentials, sent by the client application to the server [200].	68
3	Example of authorised HTTP request with a MAC access token sent by the client application to the resource server.	70
4	Example of response from the Street Identity API Discovery endpoint.	77
5	Example of response from the Street Identity API Token Info endpoint.	78
6	Example of a resource set registration request issued by a Host to AM.	129
7	Resource set registration response containing the registered set's revision and the URI of the policy.	130
8	Host registers a permission at the AM's Permission Registration endpoint.	138
9	Example of AM's RPT status response returned to the Host.	140
10	Resource registration JAX-RS endpoint for Host applications.	170
11	Example simplified code of the Request Filter component.	171
12	Registering a resource set at AM and creating a ResourceMap on a Host application.	185
13	Dynamically generating a widget for integration with Authorisation Manager.	187
14	Puma Warden logic that checks if a resource is UMA-protected (simplified code) [123].	189
15	Requester obtains an empty RPT and includes it in a request to a protected resource on a Host (simplified code) [124].	193
16	Requester processing responses from Host and AM with attached claims information (simplified code) [124].	194
17	Response from SMARTAM V1 containing information about required list of claims.	208
18	Example of a claims requested document issued by SMARTAM V1.	209
19	Example of a claims document with a self-asserted claim sent by a Requester to SMARTAM V1.	209

20	Evaluating submitted claims by the claims engine at SMARTAM V1 (simplified code).	210
21	Authorisation token response sent by AM to Requester.	210
22	Example implementation of an UMA Requester client code (simplified).	218
23	HTTP request of Requester applying for permissions to be assigned to an RPT. .	251
24	Claims requested document issued by AM.	252
25	Example UMA/j Host configuration specified in the Spring XML-based configuration metadata file.	298
26	Protecting the applications API with UMA using Puma Warden [123].	305
27	Example of Authorisation Manager Configuration Data - SMARTAM V2.	312

Chapter 1

Introduction

Identity management is an important part of computer security that requires a strong establishment of digital identities [144]. Its role is to encapsulate people, processes and services and to allow identification and management of resources that are used in any computing system. Identity management allows to authenticate subjects, such as users or processes, and to grant or deny access rights to a system's resources. Identity management comprises of *identification*, *authentication*, *access control* (also known as *authorisation*) and *auditing* [278]. Identification and authentication are used to establish a digital identity, while access control answers the question of what this identity is entitled to do. Auditing, on the other hand, allows for insight into identification, authentication, as well as authorisation events within a computing system.

Access control, being an important part of identity management, protects resources against unauthorised disclosure and unauthorised or improper modifications [277; 275]. Protected resources in computing systems may include personal information, attributes about an individual or a group of individuals, documents and photos, or any information, data¹ or services to which access should be restricted. In its simplest form, access control allows for making decisions about who (*subject*) can access what (*object*) in which way (*actions*). Therefore, it is inevitably linked with the aforementioned identification and authentication which allow for identifying subjects and verifying their credentials, respectively.

In desktop systems, end users are often in control over their resources. In such non-networked systems, not only do they host resources using a single device under their control but they also typically limit access to these resources with a single mechanism (and a single User Interface - UI) that is commonly based on the *reference monitor* concept introduced in early 1970s in [224].

¹Data and information constitute two distinct terms. Data most commonly refers to raw and unorganised facts while information refers to data which has been already processed, organised and structured in some way. In this thesis, however, there is no need for a distinction between these two terms during discussions on access control and both terms are used interchangeably.

For example, most modern multi-user operating systems keep the user data private but provide them with user interfaces that allow sharing selected data with other users or applications of the same system. Importantly, the set of subjects in these systems is limited to the users and programs available on the system, while actions are defined by the system itself (e.g. **read**, **write**, and **execute** operations available on Unix-like systems). Closed environments, such as those built on LDAP [280], Windows Active Directory [50] or internally deployed Web systems, that are used extensively within enterprises, allow to achieve a comparable functionality because of their finite and well-defined set of users, resources and possible actions on those resources.

In contrast to desktop systems and enterprise settings, the Web (and Web 2.0²) is highly distributed with its parts being under control of different authorities. Data is no longer stored in a single location (or in a set of well-defined locations) but is rather hosted on different Web applications often residing in distinct administrative domains with no existing trust relationships between each other. Commonly, these applications use a traditional isolated authorisation model where access control mechanisms are tightly bound to the application. Such mechanisms often have limited flexibility in terms of their configuration or adaptation to a particular user's security, usability or sharing requirements [203; 194].

The Web has become exceedingly user-centric and user-driven since its been first proposed by Tim Berners-Lee in 1989 [145]. In existing Web applications, it is the user who creates the data, who disseminates it and who shares it with other users and services on the Web. Interacting with Web applications, as well as creating, storing and sharing resources poses new security and usability challenges. The identification and authentication problems have been well-researched by industry and academia and have resulted in the shift of the Web from isolated technologies towards a more user-centric identity model [272; 102; 78; 58]. In such model, authentication is delegated to specialised third party components. A Web application, instead of authenticating users by itself, acts as a Relying Party (RP) that relies on the Identity Provider (IDP) to authenticate the user correctly. Importantly, users can often choose Identity Providers that satisfy their specific needs for a particular Web transaction and these users have a certain level of control over the release of their identity information to RP applications [249]. For example, some users may sign in to a social Web application (e.g. Pinterest [65]) with their Facebook account, but may use their corporate or University account to sign in to an Online Blogging Platform (e.g Wordpress [90]).

There are various technologies that allow applications to delegate authentication to third party IDPs and that support users with better control over their digital information. Most

²The term Web 2.0 was coined in the year 1999 to describe web sites that use technology beyond the static pages of earlier web sites [88; 266]. Throughout this thesis, this term is used interchangeably with the term "Web" in relation to collaborative, user-driven and distributed Web applications.

notable and widely used proposals include OpenID [104], OAuth 1.0/2.0 [200; 202], OpenID Connect [58], or various other proposed standards such as SAML [102] (implemented widely in Shibboleth [72]), ID-WSF [43], among many others. Web applications implement these proposals using libraries and frameworks. These proposals can be directly available from IDPs such as Google (Google+ Sign In [27] and Google Identity Toolkit [26] with the AccountChooser extension [263]), Facebook (using Facebook Login [11]), or the UK Federation WAYF (Where Are You From) [83]. Furthermore, these identity federation technologies can be used through "Identity Hubs" or "Identity Aggregators" with such examples as Janrain [32] or Giga [14].

Unlike authentication, access control is currently incomparable in terms of maturity, availability of technologies or software solutions for Web applications [240; 238]. In particular, existing access control is poorly addressed in the open Web environment and it is not well suited to the increasing number of distributed resources that are made available online every day. Most importantly, the modern Web still lacks a comparable access control solution based on concepts analogous to existing user-centric authentication technologies. These authentication technologies allow users to select their preferred IDPs that should be used when the user signs in to an application. As such, the user can have an account on a single IDP and may be able to sign in to all their applications that support and trust this particular IDP.

A user-centric access control solution would allow users to choose their preferred access control software components and use their functionality across a number of different Web applications, irrespectively of where these applications reside [194]. Similarly to user-centric authentication mechanisms, such access control solution would require applications to support and trust the users' preferred components. It would also support selective sharing of resources in complex online transactions with other Web users and services.

This thesis demonstrates novel approaches that satisfy requirements for user-centric access control solutions for the Web. Firstly, it presents a solution to user-driven³ access management for Web 2.0 applications and discusses its architecture and implementation. This proposal, named User-Managed Access Control (UMAC), has been introduced in [241], [243] and [244]. UMAC consists of an architecture of services and a protocol that defines interactions between these services. UMAC puts a user in full control of assigning access rights to their resources which may be spread across multiple cloud-based Web applications.

This thesis then focuses on User-Managed Access (UMA) [297], first proposed as Protect-Serve in [177] and defined in [176], which also discusses a user-centric design for managing authorisations to distributed Web resources. UMA provides a method for users to control third-party application access to their protected Web resources residing on any number of host applications.

³This thesis uses the terms "user-driven", "user-managed" and "user-centric" interchangeably. These terms are used to highlight the pivotal role of the end-user in various steps of the proposed access control solutions.

The UMA architecture and its protocol, credentials used by different UMA entities, the UMA trust model, and limitations of this proposal are presented.

UMA has been initially researched by the User-Managed Access Work Group (UMA WG) [86] at Kantara Initiative⁴ [40] virtually in parallel with our work on access control solutions for the Web. Importantly, I did not think it was useful to continue separate work on UMAC while concurrently UMA was being developed. Therefore, this dissertation focuses on the UMA proposal and a novel UMA software design and implementation, leveraging our increased user experience, design, and understanding of issues related to development of systems similar to UMAC. This thesis briefly describes similarities and differences between UMAC and UMA.

Unlike existing authorisation systems, both UMAC and UMA rely on user's centrally located security requirements⁵ for Web resources. Such security requirements can be expressed in the form of access control policies and are stored and evaluated in a specialised component called Authorisation Manager⁶. By centralising these requirements, both proposals give end user a consistent User Experience for managing access control across their Web applications. Moreover, these proposals provide users with a holistic view of the security applied to distributed data.

This thesis presents the design and implementation of UMAC. It then focuses on the UMA proposal and presents its architecture and protocol. It also demonstrates UMA implementations. Firstly, it shows UMA frameworks that allow Web applications to interact with the implemented Authorisation Manager according to the UMA protocol. Furthermore, it presents two UMA Authorisation Managers. This thesis also discusses evaluation of the User Experience of the first implemented Authorisation Manager by presenting results of the conducted user study. Moreover, it demonstrates evaluation of the UMA proposal and its implementation against the formulated set of requirements for user-managed access control solutions.

1.1 Motivation

"The goal of a flexible, user-centric identity management infrastructure must be to allow the user to quickly determine what information will be revealed to which parties and for what purposes, how trustworthy those parties are and how they will handle the information, and what the consequences of sharing their information will be." [158]

The above quote from Dr Ann Cavoukian, the Information and Privacy Commissioner of

⁴Kantara Initiative is a non-profit association dedicated to advancing technical and legal innovation in the field of digital identity management.

⁵This thesis occasionally uses the term "security requirements" to refer to "access control policies". Importantly, access control is only a subset of computer security.

⁶The UMA solution has eventually adopted a different terminology aligned with the OAuth 2.0 protocol. The most recent terminology used by UMA can be found in the most recent revision of [297].

Ontario, captures precisely the motivation behind the research presented in this thesis. In particular, this quote relates to a need for a new user-centric authorisation system for highly distributed environments such as the open Web.

Section 1.1.1 presents an example scenario with a complex Web transaction that involves secure data sharing. This scenario helps to better understand the aforementioned quote and the motivation towards a novel access management system. A thorough analysis of this scenario is provided and this analysis helps to identify shortcomings in existing authorisation solutions. These shortcomings were used to derive a set of high-level requirements which should be met by a desired user-centric access control system.

The scenario presented in Section 1.1.1 is only one of many that were published in [243] and [85]. The research presented in this thesis considers characteristics and peculiarities of various identified scenarios.

1.1.1 Scenario

A graduate student stores their academic-related information in an online database, similar to existing Personal Data Store (PDS) systems (see [64; 51; 63] for examples of such systems). This system resembles the one available at Newcastle University, called Student Self-Service Portal (S3P) [69], which allows students to access their "Graduate Programme" information and "Transcript of Records" documents. Additionally, the student has an account at the Online Courses Service, where they host a certificate for one of their completed courses (examples of such providers include popular Web sites such as Coursera [6], Udacity [82] and Mooc [49]). The student also stores their photos at the Online Gallery Service and stores their documents in the Online Storage Service. This scenario is visualised in Figure 1.1.

The aforementioned applications reside in distinct Web domains⁷ and are under control of different authorities. Importantly, the user is the author/source of authority for only some of their information (e.g. their name or their photos) while other types of information, such as "Transcript of Records" documents and certificates have other sources of authority (e.g. respective institutions).

The aforementioned applications provide Web APIs⁸ to allow programmatic access to data. The presented scenario assumes that APIs are protected using existing Web protocols, such as OAuth 1.0a [200], OAuth 2.0 [202], API keys, or HTTP Basic or Digest Authentication [192].

⁷A Web domain represents a set of Web applications within a single administrative domain that share a common set of features, including common authentication or authorisation functionality. The Web environment is discussed in Section 2.2.2.

⁸Web APIs have been the key growth driver for most applications from a wide range of industry sectors [127]. As of 28th April 2013, the Web Services Directory at ProgrammableWeb [66] lists almost 9000 Application Programming Interfaces available on the Web. Since 2005/2006, this number has been growing exponentially.

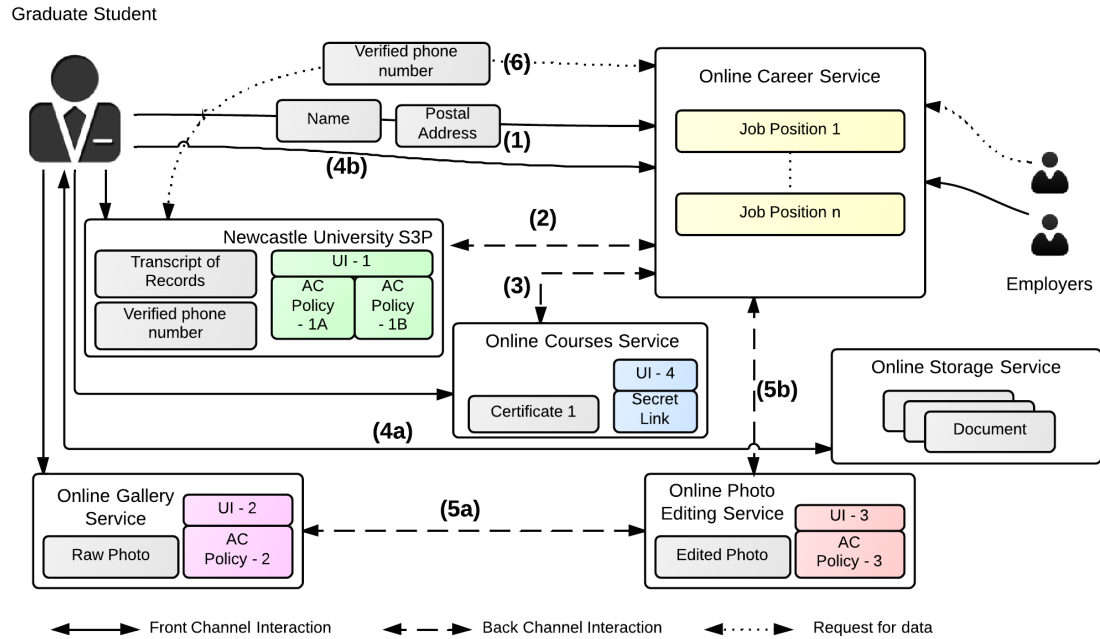


Figure 1.1: Scenario highlighting interactions and the flow of data between Web applications.

Online Courses Service uses secret links to protect access to hosted data. Such secret links are hard-to-guess URLs that uniquely identify a resource that a person wants to share.

In the presented scenario, the graduate student interacts with the Online Career Service through its provided UI. This application allows students to create their personal accounts and apply for advertised job positions. Students have to fill in an application form with the necessary details, including personal information (e.g. their verified name), "Transcript of Records" document and additional certificates from various institutions. Students may also choose to upload their photos. In the presented scenario, employers can also request additional information from students (e.g. phone number).

In order to complete the application form for "Job Position 1" at Online Career Service, the student can either fill in certain fields of the application form manually or can import the data from online systems using their Web APIs. In the presented scenario, the student completes the steps that are visualised in Figure 1.1. Each step has a corresponding number in the figure.

1. Student provides their personal information by manually filling in their name and address information;
2. The verified "Transcript of Records" document is retrieved by the Online Career Service directly from the S3P application using its Web API;
3. Student provides a secret URL to their "Certificate 1" document hosted on the Online

Courses Service;

4. Student downloads a "Certificate 2" document from their Online Storage Service and uploads it to the Online Career Service;
5. The edited photo of a student is retrieved directly from the Online Photo Editing Service using its Web API (firstly, this photo is provisioned to this application from the Online Gallery Service).

When the application form is completed, the student can submit it to finalise the first step of the job application process. The student can repeat the above steps when filling in an application form for another job position (e.g. *"Job Position 2"*).

In many cases, the application process takes time and some of the submitted information can change in the meantime. For example, the student can change their address and their "Transcript of Records" document can be updated with newly obtained marks. Importantly, such information is updated at the respective applications that store the data. Therefore, in the presented scenario, providing updated information to potential employers requires manual intervention from the student either on the Online Career Service or through direct interaction with the employer.

Potential employers, who act as receiving parties of the information from students, may decide that some information is missing and that such information should be additionally provided. Therefore, an additional step is specified in the presented scenario:

6. The employer asks the student to provide access to their verified phone number information.

1.1.2 Shortcomings

This section analyses the presented scenario. It discusses the shortcomings that can be observed in existing access control solutions. These shortcomings have not yet been addressed or have been addressed only partially by access control proposals for the Web. As discussed further in this thesis, both UMAC and UMA in particular address all of these shortcomings collectively.

- S1** Access control available at existing cloud-based Web 2.0 applications lacks sophistication and does not fit well to complex Web transactions that involve sharing multiple resources and multiple users or applications as recipients of those resources;
- S2** The user has to define their security settings for online data using diverse and possibly incompatible policy languages across distributed applications;

- S3** The user needs to use many diverse and bespoke policy management tools with diversified User Experience for managing access to their online data;
- S4** The user lacks a consolidated view of the applied access control policies and audit information across multiple Web applications.

The first weakness **S1** relates to the fact that most Web applications implement an isolated access control model where the authorisation function is tightly bound to the application. This limits its configuration or adaptation to particular user's security and usability requirements. Such access control can often only address relatively simple scenarios where data is either made public or accessible only by a predefined set of users of the application. More complex scenarios are handled by externalised authorisation components but these are only used in enterprise settings and not on the open Web. Authorisation systems focus on protecting access to Web data in single applications and during simple transactions. These systems are very limited in more complex transactions that involve sharing multiple distributed resources with multiple different recipients. Additionally, these authorisation systems are limited when it comes to sharing data with users or with services on the Web. These systems also fail to provide means for allowing recipients to apply for access in the absence of access control policies. This is related to the fact that access control simply lacks sophistication required by existing complex Web transactions (such as the one discussed in the scenario) because access control is considered a side issue for typical cloud-based Web 2.0 applications.

In the presented scenario, the student uses authorisation mechanisms provided by respective Web applications. These mechanisms may not necessarily meet all security or usability requirements of the student. These mechanisms may not allow the student to group recipients of protected data and assign access rights to such groups. In the provided example of a Web transaction, the "Transcript of Records" document is shared separately when applying for "Job Position 1" and "Job Position 2" (refer to Figure 1.1). This results in two access control policies (or two entries in a single policy) for a single resource being created and managed by the end-user at the S3P application. Surprisingly, a simple functionality of grouping users in authorisation policies is missing in numerous popular Web applications. Furthermore, sharing the "Transcript of Records" document and other data required during the job application process has to be done separately. This is despite the fact that these resources are shared as part of a single Web transaction (i.e. the user shares some data to achieve a common goal like applying for a job position). Furthermore, potential employers are unable to easily apply for access to protected resources (see step 6 of the presented scenario). Policies need to be created in advance in most existing Web applications, which does not allow to share data on "as needed" basis.

When access control does not fully meet the security or usability requirements of the user, this user may decide to share data manually, as in the case of sharing name and address information in the presented scenario. Such provisioning of data by hand and by value poses challenges as the user loses control over how data is managed by the receiving party as well as how data gets updated if necessary. This provisioning is also very inefficient. Sharing using secret links, as in the case of the discussed certificate stored on the Online Courses Service, may be perceived as convenient but it is very insecure and unsuitable for sensitive data. Links can be shared further by receiving parties without the user's consent, which means that the data can be accessed by others without the user's knowledge.

Delegating access control from Web applications to more specialised components using existing proposals and their frameworks and libraries may mitigate the aforementioned problem. In the context of Web 2.0 applications, these frameworks include the ESAPI framework from OWASP⁹ [61; 60], OAuth 1.0a, and OAuth 2.0 frameworks. Delegation in enterprise settings is often achieved using such standards as XACML [97; 109; 107; 113] and SAML [102; 98]. Moreover, there are various academic proposals for delegated authorisation. Existing access control proposals are discussed in Chapter 2 and in Chapter 3 where the shortcomings of these proposals are also presented.

Secondly, a lack of a unified policy language used among cloud-based Web applications can be observed (**S2**). These applications, being under control of different authorities, potentially use authorisation mechanisms based on distinct access control policy types based on diverse and possibly incompatible policy languages. Despite the fact that such languages are not exposed to end-users directly but only through User Interfaces, forcing the user to define their security in diverse languages becomes a problem in distributed settings. In particular, it is not easy to reapply policies when resources are moved between applications or compose a policy for a group of resources residing in numerous applications (e.g. the student cannot create a single policy for all "job application-related resources" that reside on a number of different applications and share these resource in a complex Web transaction).

In the presented scenario, the S3P application may use a simple access control matrix and the Online Gallery or Online Photo Editing services may support more flexible and expressive policy languages. Such diversity requires the student to familiarise themselves with numerous access control solutions. Additionally, the presented Online Courses Service does not implement any policies but uses secret URLs that allow access to Web data. In such cases, the student is unable to define access control rules only once and apply these rules to their personal information, "Transcript of Records" document or certificates which are spread across different applications.

⁹The Open Web Application Security Project (OWASP) is an open-source community that works to create freely-available articles, methodologies, documentation, tools, and technologies in the field of Web security.

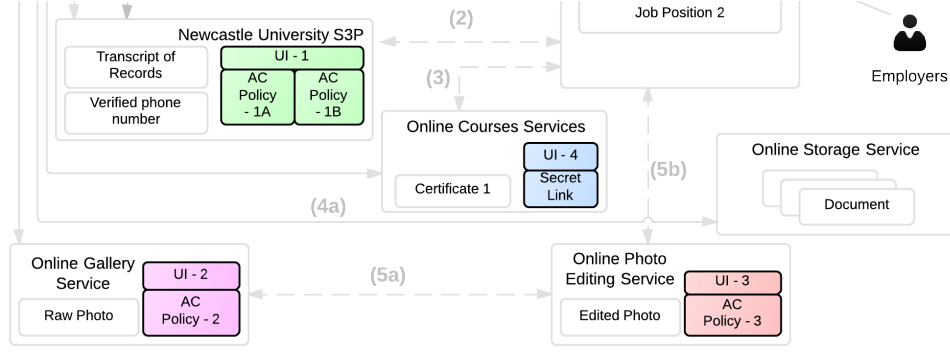


Figure 1.2: Distributed access control policies created and managed in example scenario.

Moreover, if the student decides to move some of their resources from one Web application to another, then they may not be able to reuse already defined access control rules (in this case, in the form of recipients of a secret URL) and may be challenged with composing new policies.

Another important weakness of existing authorisation solutions are custom management tools and their diverse User Interfaces used by Web applications (**S3**). This results in inconsistent User Experience (UX) when managing access to resources involved in a single transaction. In the presented scenario, the student has to concern themselves with interfaces of numerous security tools at their Web applications in order to share resources efficiently, and most importantly, securely. Despite the fact that these resources are shared as part of a single Web transaction, the student has to concern themselves with three different UIs for managing access control as well as one UI for generating secret URLs. These interfaces may differ significantly from one another with some of them being more usable than others. This weakness is visualised in Figure 1.2 that clearly shows that different UIs available need to be used at different applications (i.e. **UI-1**, **UI-2**, **UI-3**, and **UI-4** in Figure 1.2).

At present, it is more probable that a user will choose their preferred Web applications based on their functionality rather than based on their security features. However, these applications are unlikely to provide security-related UIs tailored to all their possible target user groups. Therefore, security conscious users may decide to abandon functionally rich applications if their security-related UX does not meet their needs and expectations. Additionally, forcing users to be concerned with multiple distinct access control solutions in order to complete transactions on the Web is neither efficient nor desirable. As discussed in [217], security decisions should be hidden in the user's workflow, and having a consistent UX is one of the means to achieve that.

The fourth weakness **S4** that can be observed is the heterogeneity and distribution of access control policies, which results in a user having a very limited view of the applied security controls over their distributed Web resources. Introducing new access control policies, modifying existing

ones, and auditing them is a challenging task and requires traversing Web applications and configuring access control at these applications separately. This weakness is also visualised in Figure 1.2 that shows multiple locations of security policies that were created to complete a single Web transaction.

In the presented scenario, the student does not have a holistic view over the applied access control rules for their personal information at the PDS, "Transcript of Records" document at the S3P application or their certificates at the Online Courses Service application. Furthermore, in the presented scenario the student shares their photo from Online Gallery with the Online Photo Editor, and only then with the Online Career Service. This results in two access control policies being created and stored at two different locations for the purpose of achieving a single goal of using the photo in the job application process. With the increasing amount of resources that the student may store on the Web and possibly share with potential employers (or other interested parties), managing relationships between policies and resources becomes challenging and highly error-prone.

Moreover, auditing access requests to protected resources requires retrieving such information from all Web applications involved in a particular transaction. The student, for example, has limited ability to correlate access requests to resources hosted on different applications. The student is required to access this information separately at each application if they want to audit how and when their data is being accessed. As shown in Figure 1.2, policies are stored at different systems and viewing what has been shared and how is a non-trivial task despite the fact that these resources are shared as part of a single Web transaction.

It is important to note that the scenario discussed in Section 1.1.1 was only one of many published in [243] as well as among those submitted to [85] that were considered during research presented in this thesis. Therefore, the aforementioned shortcomings **S1 - S4** can be related to other scenarios as well.

1.1.3 Requirements

This section presents the core requirements for a new access control solution, formulated based on identified shortcomings. These requirements are in line with the key trend of the Web 2.0 environment to include a user as a core part of its model. In particular, when formulating these requirements the aim was to provide a system that can be defined as following:

User-Managed Access Control System *is a centralised system that allows the user play a pivotal role in managing access to various distributed online resources and supports secure and selective sharing of those resources in complex online transactions with other Web users and*

services.

The identified high-level requirements are defined as:

- R1** The user should have a choice to use the access control mechanism and interface which provides the required sophistication, functionality and usability as well as satisfies their needs for their Web applications to allow them share their data securely and selectively with other users and services on the Web;
- R2** The user should be able to compose access control policies for their Web resources in a uniform way and in a single policy language of their choice and the user should be able to apply these policies to distributed and heterogeneous data;
- R3** The user should be able to compose access control policies using a single policy management tool that would provide a consistent User Experience when assigning access rights to Web resources hosted at distributed Web applications that reside in arbitrary Web domains;
- R4** A consolidated view of the applied security controls as well as audit information should be provided to support users with managing access to the ever-growing number of resources on the Web.

The above mentioned high-level requirements aim to address the previously described shortcomings, i.e. requirement R1 addresses the shortcoming S1 and so forth. Furthermore, these requirements can be mapped directly to the quote from Dr Ann Cavoukian that is presented in Section 1.1.

The defined requirements were used to provide a user with the necessary solution to manage security for distributed resources on the Web. Importantly, such solution would be able to provide access control to resources which the user owns (i.e. where the user is the source of authority) as well as resources that the user controls (i.e. where the source of authority is different from the user, e.g. it is an organisation).

1.2 Goal

The research on user-driven access control solutions for the open Web aims to make a step forward to empowering individuals with control over access to their online data. Privacy is about allowing individuals to possess the means and tools to share their personal information and other resources selectively to achieve desired goals and securely complete various transactions on the Web. Therefore, modern access control solutions should not focus on allowing users to keep their data inaccessible but should support such selective, flexible and secure sharing.

Goal: *The primary aim is to define a secure and thorough proposal for a user-driven access control system that meets formulated requirements and allows individuals to have full control over access to their distributed Web resources.*

Objectives: *In order to achieve the presented goal, it is necessary to define the architecture and the protocol for a user-centric access control proposal. It is also necessary to provide a software implementation of such proposal and share knowledge and practical experience gathered during its development. The provided system should prove to be both user as well as developer friendly, as both characteristics are crucial in achieving the uptake of any security solution. Furthermore, the implemented system should support integration with new and existing applications, through the use of developed frameworks. Finally, findings of the conducted research should be applied to other existing authorisation solutions. This would allow to extend the user-driven approach to managing access to distributed data beyond the original proposal.*

1.3 Approach

The goal of providing a user-managed access control system for the Web has been achieved through an iterative design and development process (Figure 1.4).

Firstly, it was necessary to study the literature in the field of identity and access management as well as existing access control systems. The focus was put on the concepts of access control in distributed environments, access control policies, delegated authorisation, as well as challenges in authorisation policies and architectures. Parts of this work were published in [240].

Furthermore, a set of scenarios that describe different types of user transactions on the Web, which are currently poorly addressed by existing authorisation solutions, was defined and published in [243]. Scenarios were analysed and used to identify shortcomings which were not addressed or were addressed only partially in existing literature and available solutions. Identified shortcomings were used to formulate requirements for a user-managed access control proposal.

An attempt was made to map the existing XACML-like solutions [97] to the Web 2.0 environment. A proposal was defined based on a simple access control query/response protocol and it is discussed in [241]. A prototype solution of this proposal was implemented. This proposal was then refined and named User-Managed Access Control¹⁰ (UMAC), which was discussed in [244]. In the prototype implementation of the refined UMAC system, a user could compose authorisation policies using a central application and apply these policies to distributed Web resources. The prototype application provided a User Interface for managing policies and a

¹⁰The original proposal was in fact called User-Centric Access Control (UCAC) and this name is used in initial publications.

RESTful API for client applications that host the data or applications that access the data.

Further research on access control has been done as part of the User-Managed Access Work Group (UMA WG) at Kantara Initiative where a proposal for a similar solution has emerged, named User Managed Access (UMA). This thesis focuses on UMA, leveraging our increased user experience, design, and understanding of issues related to implementation of systems similar to UMAC. Importantly, I have contributed to the work group, including contributing findings from our own design and implementation. I have been a member of the UMA work group since 2009. Since 2010, I have been the Vice-Chair of the Work Group as well as the Use Cases Editor and the Implementation Coordinator. I am also the co-author of the UMA protocol specification. The list of members of the User-Managed Access Work Group is maintained at [87].

An UMA proposal was implemented during the SMART (Student-Managed Access to Online Resource) project [77]. The developed UMA implementation consisted of various frameworks, namely OAuth Leeloo, UMA/j and Puma. This software was used to build client applications that were used to evaluate the UMA proposal. It was also used to get experience in integrating access control features with new and existing Web applications.

The UMA implementation also included Authorisation Manager applications. Initially, a prototype Authorisation Manager, called SMARTAM, was developed and this system was evaluated using the conducted user study. Based on this study, a second version of SMARTAM implementation was provided. This version addresses numerous usability issues that were identified in the study. It also provides additional functionality in comparison to the first version.

Moreover, our knowledge and experience was contributed to the Google's emerging Street Identity proposal [28], which aimed to allow selective access to distributed user attributes. Such hands-on experience in designing, developing and deploying a highly scalable and robust identity and access management protocol allowed us to further refine some of the features in our own design and implementation.

1.4 Novelty of Approach

Both proposals for access control for the Web 2.0 environment, which are discussed in this thesis, have several advantages over existing authorisation solutions. The presented proposals address the identified shortcomings and meet formulated requirements. Both proposals also fit precisely to the user-driven Web 2.0 environment. This section focuses primarily on presenting the novelty of the UMA proposal and the UMA implementation, which are discussed in this thesis.

Firstly, in the UMA proposal, access control functionality is separated from Web applications

and is provided in the form of a pluggable Web service, named Authorisation Manager¹¹ (AM). Access control may vary depending on the particular AM used by the end-user and does not depend on the actual Web application. Therefore, the user can decide to use the AM that satisfies their particular needs and provides the required level of security, usability and functionality. With such approach, different groups of users will be able to benefit from the functionality provided by a particular Web application and will be able to choose the AM which precisely satisfies their needs. In particular, the AM is able to provide access control functions necessary to satisfy common Web transactions that involve sharing resources from multiple Web applications, such as those presented in Section 1.1.

Additionally, unlike existing systems, the user of the UMA system can compose access control policies for many or all of their applications using a single policy language that is supported by the user's preferred AM. As such, in this proposal the user is able to easily define and apply a single policy to resources hosted on different and distributed Web applications. UMA does not require the AM to understand the representations of resources that it protects. Its functionality is applicable to arbitrary resources which can be identified with URLs. It has been shown that this proposal can be applied to such data as photos and documents as well as individual pieces of information and attributes (details are discussed in further chapters of this thesis). Moreover, reusing a policy can allow a user to set up security before actually sharing a particular resource at any application, which is a part of the UMA proposal. The above mentioned functionality is not present in existing proposals or solutions on the Web.

In the proposals discussed in this thesis, a user can apply a single policy across a set of distributed Web resources and is given a unified User Experience throughout the entire process of composing an access control policy, linking this policy with resources, as well as amending this policy as necessary. Therefore, a user does not have to interact with different authorisation solutions in order to complete a single Web transaction that may involve distributed resources owned or controlled by this user.

With a central component responsible for access control, a user has a holistic view of the applied security controls for their Web resources. Importantly, a user can introduce new policies, modify existing ones, and audit them from a single location. Access requests to resources on different hosts are evaluated centrally by AM. Therefore, a user may easily audit various aspects of such requests and correlate them as necessary without the need to pull logging information from all hosts involved in a Web transaction.

A user is also not required to be directly involved in interactions between services accessing

¹¹The name Authorisation Manager will be introduced and defined in further chapters of this thesis and this term is used to describe a specialised component concerned with making access control decisions for distributed Web resources.

data and services hosting data after such user successfully sets up policies at AM. The necessity of such direct involvement is one of the shortcomings of existing proposals (e.g. OAuth 1.0a, OAuth 2.0 or OpenID Connect). In UMA, various interactions between distributed applications are controlled in a fashion guided by a user's defined policies that are managed and audited through the digital "footprint dashboard" provided by AM.

1.5 Contributions

The main contributions of the PhD research are listed below. Importantly, there were others who were involved in the work on some of the listed contributions. Therefore, this section should be read along with the *Acknowledgements* section given earlier that provides more details on the involvement of others in the conducted research.

1. A thorough analysis of existing access control proposals for distributed computing environments;
2. Architecture and protocol for the user-controlled access management solution for distributed environments (User-Managed Access Control);
3. Contribution to the design and development of User-Managed Access at Kantara Initiative;
4. Software design for UMAC and UMA proposals;
5. Comprehensive software implementation, including:
 - (a) User-Managed Access Authorisation Manager, with various extensions, as discussed in this thesis;
 - (b) Java framework for building UMA-enabled Web applications;
 - (c) Python framework for building UMA-enabled Web applications;
 - (d) One of the first complete OAuth 2.0 frameworks for Java applications;
6. Policy model for the implemented UMA Authorisation Manager;
7. User studies of the proposed UMA Authorisation Manager;
8. Contribution to the design and development of the Google's Street Identity protocol, including multiple prototype implementations.

A thorough analysis of existing access control proposals for distributed computing environments was used to identify shortcomings in these proposals. In particular, proposals for

multi-domain computing environments, the open Web and Web 2.0 applications were reviewed. The focus was put on the concepts of access control in distributed environments, access control policies, delegated authorisation, as well as challenges in authorisation policies and architectures.

A scenario and use case analysis, which was originally published in [243], was then provided. It was necessary to analyse various Web transactions to identify weaknesses of existing authorisation solutions. An extensive literature review of available authorisation solutions was also conducted. Based on this analysis and review, a set of high-level requirements for a new access control proposal was formulated. As discussed in this thesis, some of the existing authorisation proposals meet either none or only selected requirements.

A new approach to user-controlled access management for the Web was designed. This proposal was named User-Managed Access Control (UMAC). UMAC puts a user in full control over access to their resources and relies on a user's centrally located security requirements for these resource, which may be scattered across multiple Web applications.

The focus was then shifted to the proposal from the User-Managed Access Work Group and further research efforts continued solely on the User-Managed Access solution (and not on UMAC) at Kantara Initiative. In collaboration with the research team at Newcastle University, I contributed to the design and development of UMA through participation and work in the UMA WG. We also contributed with the knowledge from the design of the software that would realise the architecture and the protocol of the UMA proposal. In particular, we implemented all parties of this proposal, including Authorisation Manager, Host and Requester applications that UMA defines.

During conducted research, an UMA Authorisation Manager called SMARTAM V1 was developed. This AM provides a UI that allows users to manage access control for distributed Web resources. It also provides a RESTful API for client applications. This contribution was awarded the Identity Deployment of Year (IDDY) Award [121] in 2011.

The UI of SMARTAM V1 was evaluated with a user study that allowed to verify the approach to user-managed access control and the developed software. Results of this evaluation were used during development of a second version of UMA Authorisation Manager called SMARTAM V2.

SMARTAM V2 included feedback gathered during the conducted user study. It also implemented a more recent revision of the UMA proposal. SMARTAM V2 was built in a modular way using developed frameworks, namely OAuth Leeloo and UMA/j.

Both developed Authorisation Managers implement a rich access control policy model. SMARTAM V1 allows users to define policies that include identities of users and optional required self-asserted claims. A policy evaluation engine to support those policies was designed and implemented as well. On the other hand, SMARTAM V2 allows users to define *myself*- and

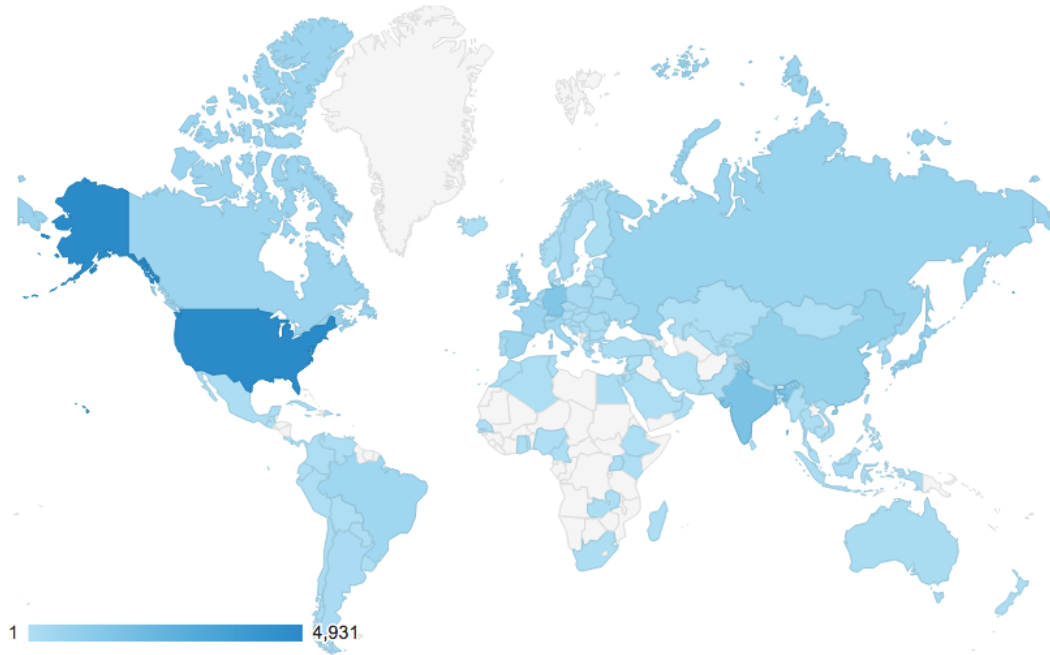


Figure 1.3: World map showing visitor demographics of the OAuth Leeloo website at [117] between 1st August 2010 and 29th April 2013. Map has been generated using the Google Analytics software [18].

others-type policies. The first type allows for selective sharing of resources with other services on the Web on behalf of the owner (or controller) of those resources. The second type of policies allows for selective sharing of resources with other users on the Web. Others-type policies can include third-party asserted claims using OpenID Connect.

In parallel to AM implementations, various frameworks related to UMA have been developed. Firstly, a complete OAuth 2.0 Java library was designed and implemented. OAuth 2.0 is the main building block of UMA and it was necessary to provide its implementation that could be easily used by other UMA software. The implemented library, named OAuth Leeloo [117], included some proposed extensions, with dynamic registration as an example. It was first released in August 2010 and submitted to the Apache Software Foundation few months later. Despite the fact that the code was incorporated into a Top-Level Project (Apache Oltu) [129], the original release of OAuth Leeloo continued to attract various communities. Between its release and the 29th April 2013, the website at [117], has been visited 20,128 times (with 11,166 unique visitors) from 117 different countries (Figure 1.3). The software has been downloaded more than 3000 times from the website, which does not include downloads from the Maven Central Repository.

OAuth Leeloo was used to develop an UMA framework in Java called UMA/j. UMA/j supported development of Authorisation Managers and UMA-enabled client applications. The

framework introduced various architectural choices that are discussed in this thesis. Additionally, its API was used to provide feedback to the UMA WG on the UMA protocol and how the protocol can be implemented in new or existing Web applications.

To further simplify development of client applications, a Python implementation of the UMA protocol was also provided with various extensions that are discussed in this thesis. Puma targets rapid application development using the popular Google's Platform-as-a-Service (PaaS) - Google App Engine [21]. Puma was used to implement one of the first UMA-enabled PDS-like systems, which is discussed in this thesis. On 26th March 2012, I was a member of the Technical Workgroup at the World Economic Forum "Re-Thinking Personal Data" - Tiger Team meeting held in San Jose, CA, USA. The meeting discussed the emerging Personal Data market and applicability of various technologies. I discussed the UMA proposal and the implemented UMA software during this meeting.

Findings from the design and development of UMAC and UMA solutions were additionally contributed to the Google's Street Identity protocol in 2011 and 2012. Street Identity allows clients, called Relying Parties (RP), to obtain verified attributes about individual users from applications called Attribute Providers (AP). Access is authorised by Google IDP that records the user's consent to share a specific attribute type and issues authorisations to RPs. Street Identity in its current form can be considered a simplified version of the UMA proposal. Importantly, I contributed throughout the entire lifecycle of the protocol, starting from design up to its deployment to the Google's production infrastructure.

1.6 Thesis Outline

This thesis is organised as following. Firstly, this chapter introduces the thesis. It outlines the main goals and the approach taken in satisfying these goals. It also introduces the motivation behind a new access control solution for distributed and user-driven environments such as Web 2.0. It presents an example scenario describing a transaction on the Web. It provides analysis of this scenario and points out the shortcomings of existing access control systems. It then presents formulated requirements for a new access control proposal. This chapter summarises the novelty of the solutions presented in this thesis and also discusses main contributions.

Access control in distributed environments is presented in Chapter 2. This chapter provides background information regarding concepts that are used throughout this thesis. It starts with an overview of multi-domain computing environments, the open Web and Web 2.0 applications, and presents their specific characteristics that are of interest in the context of access control. Furthermore, it discusses different concepts of access control in distributed environments, access

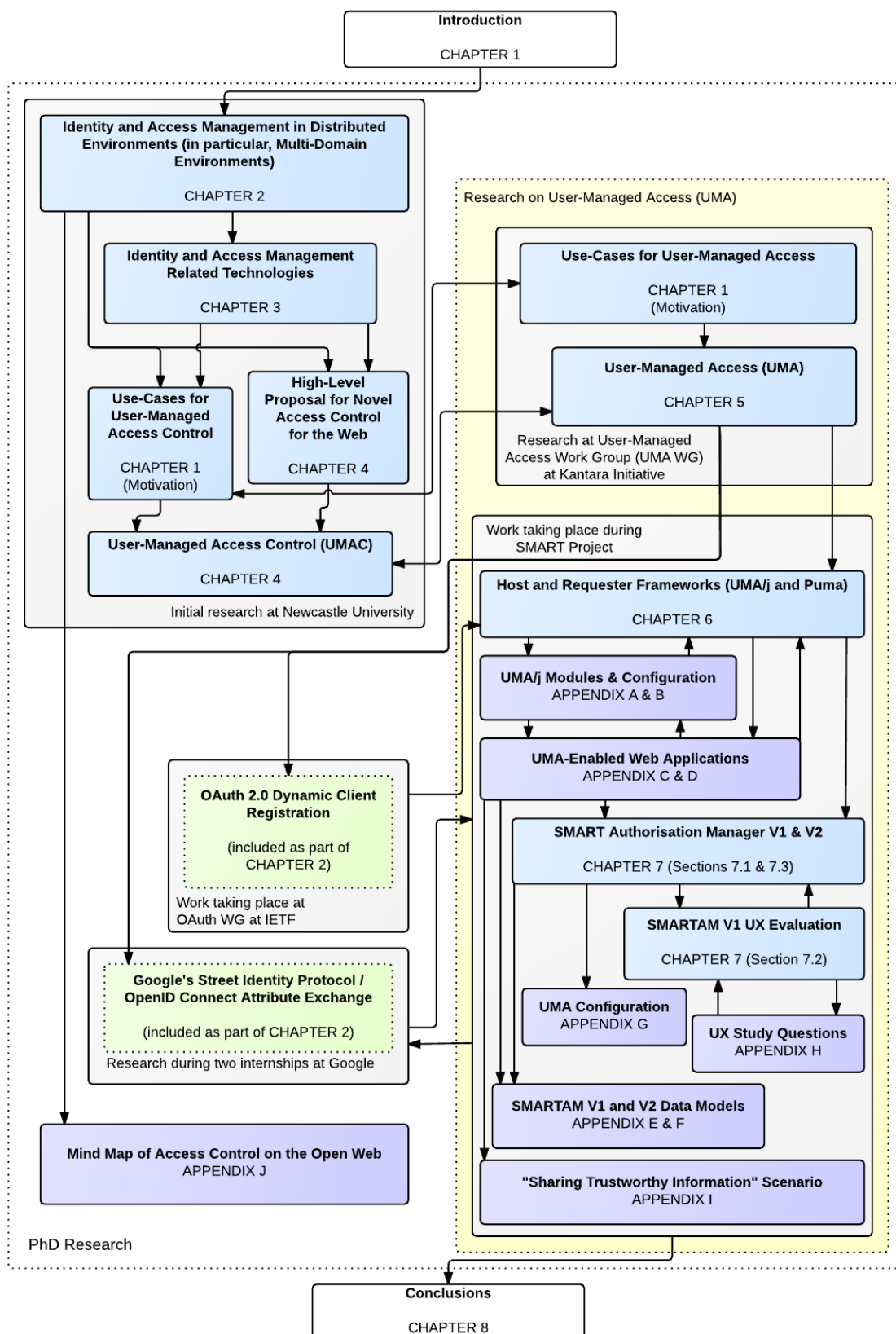


Figure 1.4: Visualisation of the thesis outline.

control policies as well as delegated authorisation. It then discusses challenges in access control policies and architectures. Parts of this chapter were first published in [240].

Chapter 3 provides a literature review. It discusses existing standards and standard proposals as well as academic work. It presents various solutions for access control that allow individuals to be in control of access to their Web data.

Chapter 4 introduces a novel authorisation solution for distributed Web resources, called User-Managed Access Control (UMAC). It discusses the architecture and the protocol of this proposal. It also shows how this proposal meets identified requirements for a new user-managed access control system. It then discusses the identified shortcomings and limitations of UMAC. Parts of this chapter were published in [242; 244].

Chapter 5 presents the User-Managed Access (UMA) solution, which is researched by the User-Managed Access Work Group at Kantara Initiative. UMA has been used in research presented in this thesis. This chapter discusses additional requirements for UMA. It also presents the UMA architecture and the protocol. This chapter then evaluates UMA and provides an overview of its limitations. It also shows differences between UMA and the UMAC approach. Importantly, Chapter 5 is based on specifications and other work published by the User-Managed Access Work Group. Furthermore, parts of this chapter were published in [238].

Chapter 6 presents two UMA frameworks that allow applications to externalise their access control functionality and act as Hosts and Requesters, as defined by the UMA protocol. Section 6.2 presents a Java implementation named UMA/j. Section 6.3 presents a Python implementation named Puma. This chapter also discusses identified limitations of both frameworks.

UMA/j was targeted at enterprise-level (but still user-driven) cloud-based Web applications. The framework provides a high-level *UMA API* for applications as well as allows to use low-level APIs for finer control of the protocol flows. Parts of Chapter 6 that discuss UMA/j were published in [239], as a joint effort with Łukasz Moreń.

PUMA, on the other hand, was built primarily for applications running on the Google App Engine PaaS [21]. Because this framework was designed and implemented later than UMA/j, it contained various enhancements and simplifications for application developers. It also implemented a more recent revision of the UMA protocol. Parts of Chapter 6 that describe Puma were first published as blog posts during the SMART project in [123] and [124], as a joint effort with Jacek Szpot.

Chapter 7 presents two UMA Authorisation Manager implementations that were developed in parallel to the work on UMA frameworks. It starts by discussing an initial AM implementation, called SMARTAM V1. It presents the design of this software and its policy model. It also discusses integration of SMARTAM V1 with example applications.

Chapter 7 then presents the conducted user study that was used for evaluation of SMARTAM V1 and its User Interface. It presents identified shortcomings of SMARTAM V1 based on the gathered feedback from the user study. It also presents requirements that were derived from these shortcomings and that were used in further work on UMA software. The user study presented in this chapter is a joint effort with MSc student Iain Carter also published in an MSc thesis in [153], where I was responsible for the design of this study.

Moreover, Chapter 7 discusses a second AM implementation, named SMARTAM V2. SMARTAM V2 addresses various shortcomings that were identified in the earlier implementation. It also provides support for a more recent revision of the UMA protocol. This chapter discusses the architecture of this software and its policy model. It also presents its unique features that provide additional value on top of the core UMA proposal. This chapter also provides an overview of shortcomings of SMARTAM V2.

Finally, Chapter 8 concludes this work. It discusses a summary of the thesis. It also presents future research directions and finally provides concluding remarks.

Chapter 2

Background

2.1 Introduction

Modern computing systems are no longer built from monolithic applications provisioned from a single and central location but rely on loosely coupled components distributed among different hosts or deployed as individual Web applications/services or even entire Web ecosystems. The Service Oriented Architecture (SOA) paradigm supports such a decentralised computing model. Software is viewed as composite applications built from single units called services that can be reused across multiple applications [175]. SOA promotes reusability of services and eases application management and integration. Using SOA principles allows to build systems that are scalable and can evolve according to changing requirements [99].

SOA provides a solid foundation for business agility as well as adaptability. Therefore, it has gained much attention during the last decade, primarily in the enterprise settings. Services are built, exposed and consumed by internal applications in intra-organisational scenarios and exposed to other domains of business partners or specialised Service Providers [215]. Systems that span multiple administrative domains form multi-domain computing environments and are created to enhance collaboration between the parties as well as to promote sharing of resources and services.

Similarly, in today's rapidly developing and open Web environment, various applications and services are provided in the Software-as-a-Service (SaaS) model. These SaaS applications provide users with functionality as varied and complex as desktop applications that would normally be used by individual users or internally within organisations. Those applications are collectively referred to as "Web 2.0", with consumer examples such as the Wordpress platform [90], Blogger [130], Google+ [16], Facebook [9], as well as business-oriented examples such as Google Apps [22],

Salesforce [68], Liferay [44], or Concur [5], among others. Those applications allow individual users or entire organisations to create, manage and share their content online. In an environment of cloud-based applications, more and more data is released online and stored in a distributed manner. Both, in multi-domain computing environments as well as on the Web, data can be accessed using such technologies as Web Services.

Web Service technology is a flexible distributed computing abstraction that enables application composition from resources and services located in different organisational or administrative domains or over the Web [94; 226]. Web Services fit well into SOA because they allow applications and resources to be exposed as services that can be accessed remotely [137]. In modern computing environments, either SOAP-based or RESTful Web Service technologies are used. The first type uses the eXtensible Markup Language (XML) [110] to describe service interfaces in WSDL [165] and to encode messages that are exchanged in SOAP [294], while the latter one, initially described in [187], is a more lightweight and scalable approach based entirely on the HTTP protocol, which follows the principles of Representational State Transfer (REST). REST principles are discussed in [187].

RESTful services, in particular, fit well into the open Web environment because of their nature of stateless operations, caching capabilities, and low-overhead messages, among other factors. These services have been recently used to build various Web-accessible Application Programming Interfaces (Web APIs), abandoning XML in favour of a more lightweight Javascript Object Notation (JSON) [166; 167]. This thesis explains how the discussed UMA proposal uses REST in its architecture and protocol and how it provides a solid authorisation solution for protection of RESTful Web services.

Web Service-based integration of computing environments from autonomous administrative domains poses numerous security challenges and mechanisms typically used for protecting resources and services within single organisations may not be sufficient. Different organisations often have conflicting security requirements [142] but they may still need to interoperate by exposing a part of their resources and services to other parties. Therefore, architecting security mechanisms for such multi-domain environments has become an important, but complex and error prone task that requires a deep understanding of the functional and non-functional requirements of each participating domain [173].

The same applies to the Web, where autonomous applications or even entire Web ecosystems may be required to share various data, according to the needs of organisations or, more importantly, individual users involved in online transactions¹. Therefore, access control, in particular, needs to be well understood to enable efficient and secure collaboration and resource sharing.

¹ This thesis focuses primarily not on the organisation but on the user who owns the data or controls the data.

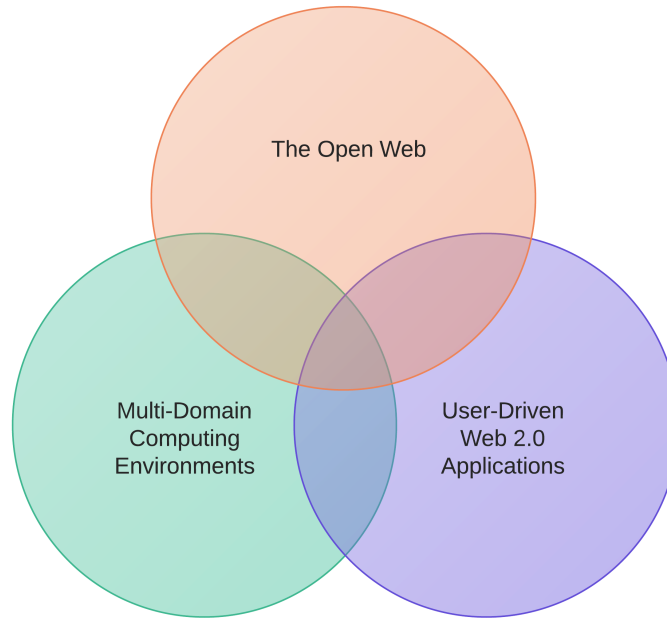


Figure 2.1: Overlap of access control requirements in different deployment environments.

Resources and services need to be protected within their administrative boundaries but should be accessible according to security requirements imposed beyond these boundaries.

This chapter provides background information regarding concepts that are used throughout this thesis and it is organised as following. Section 2.2 introduces distributed computing environments, including multi-domain computing environments, the open Web and Web 2.0 applications. It presents their specific characteristics that are of interest in the context of access control. As shown in Figure 2.1, requirements for access control for the Web and Web 2.0 overlap with the requirements of multi-domain computing environments. Such overlap has been recognised in existing literature, e.g. [281], which presents parallels with the issues of data sharing and trust in Social Network applications and those that have arisen in the e-* (e.g. e-Science, e-Research, e-Health, e-Business, etc.) arena. Section 2.3 presents access control mechanisms and discusses them in the context of distributed computing environments. It also introduces delegated authorisation and presents its various derivatives. Section 2.6 discusses access control policy and architecture challenges.

2.2 Distributed Environments

This section introduces distributed environments and provides a brief overview of their architectures.

2.2.1 Multi-Domain Computing Environments

Multi-domain computing environments have evolved to provide means of resource sharing and problem solving among various institutions [189]. These environments are composed of multiple separate and autonomous administrative domains where each domain belongs to a different entity. For example, domains can be managed by single users, departments within a company or most typically entire organisations. Such environments may run on collaborations in an ad-hoc fashion [228]. Those are typically peer-to-peer based bilateral collaborations where partners do not need to have pre-established trust relationships. Collaborations between multiple domains can be also in the form of federated environments. Such environments are designed to simulate a similar environment to a single domain with pre-established trust relationships between all collaborating partners.

A multi-domain computing environment, when composed to address a specific business or science related problem, is often referred to as a *Virtual Organisation (VO)*, as described in [189]. Such environment supports collaboration between parties of specific expertise. A high-level view of a multi-domain computing environment in the form of a Virtual Organisation is depicted in Figure 2.2. Each VO member has its own set of resources (*WS*) protected by policy enforcement points (*PEP*). Policies are stored by administration points (*PAP*) and are evaluated by decision points (*PDP*). Policy enforcement, decision and administration points constitute the building blocks of a general policy-based authorisation system and are discussed in Section 2.4. These components have been used as building blocks in the proposals that are discussed in this thesis.

Organisations that form VOs have trust relationships established between each other. In Figure 2.2, *Organisation 1* trusts *Organisation 2* (T_{O12}), while *Organisation 2* trusts *Organisation 3* (T_{O23}). In the provided example, trust is bidirectional between respective systems but is not transitive, which means that *Organisations 1* and *3* do not necessarily have to trust each other despite that T_{O12} and T_{O23} are in place.

Individuals and institutions that form a VO share data and applications implemented as resources or services. Due to the highly distributed nature of shared resources/services and a limited trust between collaborating partners such sharing needs to be controlled. Providers of resources need to be able to define clearly and carefully how these resources are provisioned, who is allowed to access those resources and what are the conditions under which access may occur. Additionally, sharing relationships can vary dynamically over time [189; 164]. This includes resources involved in sharing, participants of the VO who want to access those resources, and access control rules that exist.

There are different types of multi-domain computing environments. They can differ in the purpose of being established, their scope, size, duration, structure, community and sociology

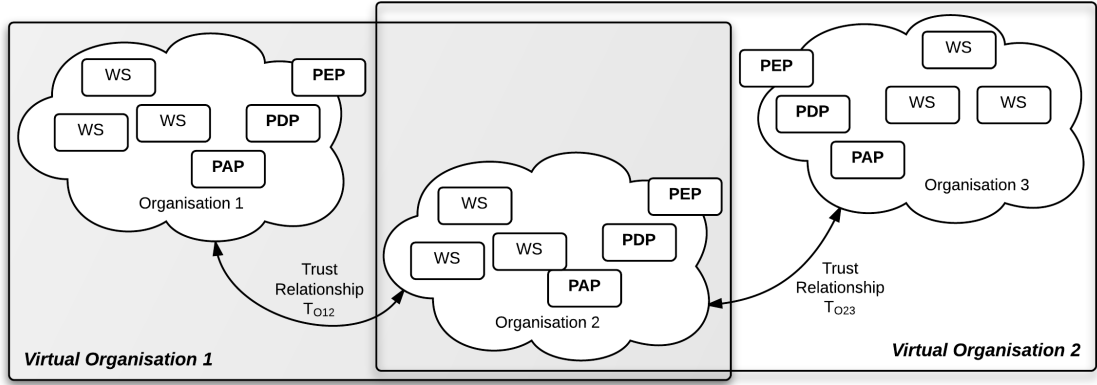


Figure 2.2: Multi-domain computing environment forming a Virtual Organisation [219].

[189]. Those characteristics influence authorisation systems that need to be employed by such environments. An example would be where the size of a multi-domain computing environment determines the distribution of components of the access control mechanism and has impact on performance of the protocols used for authorisation. Duration of such environment may also influence the way permissions are assigned to entities. For example, in highly dynamic and large environments, access control can be based not on explicitly named set of individuals but on the attributes of those individuals. For example, access control policies can contain rules for participants with certain attributes, capabilities, or levels of trust, among others, rather than for those that have specific identity credentials or even roles.

Existing multi-domain environments, such as Virtual Organisations, often have a well-defined set of access control components (e.g. policy decision and administration points) and are commonly concerned with: (1) enforcing access control policies for inter-domain access requests, (2) enforcing access control policies within individual domains. Interactions between these components are not necessarily established dynamically as and when required but they are established during formation of a VO. Resources and services that are protected with a policy-based authorisation system refer to decision points based on existing configurations. Therefore, it is often the responsibility of a VO administrator (or a set of administrators) to establish relationships between services and access control components and to define security policies for resources and services. The user is not directly involved in these processes but may be able to compose policies for resources and services that this user owns or controls. As discussed in this thesis, excluding the user from access control processes is considered as one of the main disadvantages of existing authorisation solutions, which does not fit well with the requirements for a user-managed access control solution for the open Web (recall requirements presented in Section 1.1.3).

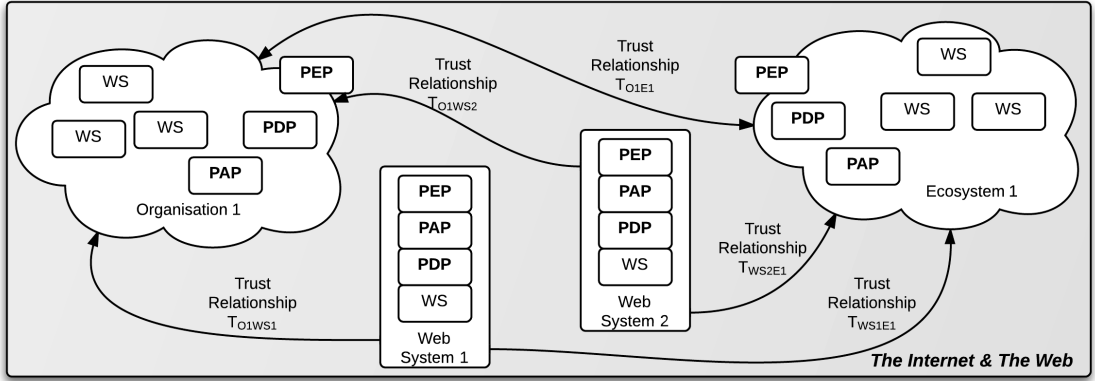


Figure 2.3: The Internet & the Web environment.

2.2.2 Web Environment

Similarly to multi-domain computing environments, the Web is composed of various distributed systems, including organisations, individual Web systems and recently emerging Web ecosystems (Figure 2.3). These systems reside in distinct domains under the control of different entities. In the first case, organisations expose their resources and services through publicly facing Web sites or Web Application Programming Interfaces (Web APIs). Individual Web systems are deployed and managed outside the perimeter of any organisation. In case of Web ecosystems, these constitute sets of various Web applications within a single administrative domain (or a set of such domains) and share a common set of features².

In the current Web model, named Web 2.0 [88; 266], these distributed systems are inherently interconnected since the Web itself enables creation of new applications by reusing and combining different services or data from various sources [251]. These services and resources are accessed using SOAP-based APIs or more recently RESTful APIs [66]. On the Web, the user is both a consumer and a provider of various distributed resources and often maintains a separate copy of identity, social relationships or access control policies at different systems [286]. As depicted in Figure 2.3, distributed resources on the Web are protected by their respective mechanisms, similarly to previously defined multi-domain computing environments. Organisational resources, despite being exposed on the Web, use the existing organisational mechanisms based on the PEP/PDP/PAP architectures, while individual systems or Web ecosystems use their own mechanisms.

On the Web, distributed systems can form trust relationships between each other, which is similar to multi-domain computing environments. However, such relationships are rarely

²An example of such a Web ecosystem is the one provided by Google, with distinct applications such as Gmail [15], Google Drive [25], YouTube [92], among many others, that share a common set of features, including authentication, authorisation, or even APIs.

pre-established because of the number of possible components. Instead, such relationships are formed in a dynamic way either by administrators of applications or (more often) by individual users of those applications. Trust relationships most commonly refer to federated authentication, where systems rely on authentication information asserted by identity providers. In Figure 2.3, *Organisation 1* trusts *Ecosystem 1* with authentication information, while *Ecosystem 1* trusts authentication from *Organisation 1* (T_{OIE1}). Additionally, *Web System 1* trusts users from *Ecosystem 1*, however, *Ecosystem 1* does not trust authentication information asserted by *Web System 1*. The latter case shows a uni-directional trust, which is more common on the open Web than bi-directional trust. In this figure, arrows denote the direction of trust. In case of uni-directional trust relationships the arrow is directed from the trustee to the trusted system.

In the context of authorisation, trust relationships allow one system to enforce access control decisions made by another system. In particular, this functionality exists for API management on the Web. Examples of such functionality include Mashery [46], 3scale [1], and Apigee [2], that differ in their architecture. However, these systems are mostly concerned with protecting organisational Web APIs that can be accessed by client applications. Therefore, these systems do not focus on giving end users the control over data that can be spread across numerous Web applications (either data owned by the user or data controlled by the user). Moreover, in these systems, there is rarely a distinction between the actual service and the owner of that service in relation to access control.

Giving control over resources and services to end users is the subject of this thesis, which discusses proposals that allow for dynamically established bi-directional trust relationships to be formed by components of the system. Moreover, in the discussed proposals, different users can delegate authorisation from the same application to different access control components depending on the requirements of these users.

2.3 Access Control in Distributed Environments

Access control (authorisation) constitutes an important part of Identity and Access Management. IAM encapsulates people, processes and services to identify and manage resources used in various information systems and this is required to authenticate users (subjects) and grant or deny access rights to specific resources and services. IAM itself comprises of the following four components:

1. Identification;
2. Authentication;
3. Access Control (Authorisation);

4. Auditing.

Identification allows users or applications to present their digital identity to the receiving party, while *Authentication* allows the receiving party to verify the correctness of such identity (e.g. by verifying credentials). *Access Control* allows to make decisions about which resource(s) a user or an application can use and what actions can be performed on these resources. *Auditing* further enhances the aforementioned processes by logging identification, authentication and authorisation events. This thesis focuses primarily on *Authorisation* and this concept is presented further in this chapter.

Access control protects resources against unauthorised disclosure and unauthorised or improper modifications. It also ensures that every access is controlled and that only authorised access requests can succeed [277; 275]. In single host computing systems, access control can be provided by establishing a barrier around this host and by making sure that all access requests are analysed and checked. For example, requests have to pass through a *reference monitor* that makes access control decisions [224] (recall Chapter 1).

In multi-host, distributed and dynamic computing environments a more flexible authorisation architecture is required. In such environments, creating a simple barrier around a group of distributed services is neither feasible nor desirable. Instead, a logical rather than a physical barrier has to be established.

Access control can be discussed at different levels of abstraction. This includes *access control policies*, *mechanisms* and *models* [275; 184]. *Policies* describe how access is managed and who, under what circumstances, may access which resources. *Mechanisms* enforce policies and define how access requests are evaluated against those policies. *Models* bridge the gap between high-level policies and low-level mechanisms by defining means of how access control rules should be applied to protect resources. Such models are defined mostly in terms of subjects and objects and possible interactions between them [184]. This thesis presents novel proposals that intervene all three levels of abstractions, i.e. policies, models and mechanisms.

2.3.1 Policies

Different access control policies have been proposed over the years. These policies, as discussed in [275], fall into one of the following 3 main categories:

1. Mandatory Policies;
2. Discretionary Policies;
3. Role-Based Policies.

In *Discretionary Access Control (DAC)* policies, access is based on the identity of the subject and on specific access control rules that define which operations are allowed on what objects (and possibly including under what conditions). *Mandatory Access Control (MAC)* policies control access based on centrally mandated sensitivity levels (classifications) of protected resources and authorisation levels of subjects (clearances). The third category, *Role-Based Access Control (RBAC)*, allows composing access control policies that map naturally to an organisation's structure [276; 275]. In RBAC, access control decisions are made based on roles that individual subjects may possess and rules that are applied to resources. RBAC merges the flexibility of explicit authorisations with additionally imposed organisational constraints. It has been further extended as *Attribute-Based Access Control (ABAC)* or *Claims-Based Access Control (CBAC)* [133; 216] to support control that is based on arbitrary types of attributes or claims. As such, RBAC and ABAC/CBAC are well suited to distributed environments that need to address security requirements for a large base of subjects and objects. Those environments include previously discussed multi-domain computing environments as well as the open Web (further explanation of the use of CBAC is given in Chapter 5).

Access control policies may contain positive statements (*positive authorisations*) or negative ones (*negative authorisations*). Statements can be presented in the form of *Access Control Lists (ACL)*. Positive statements define what actions can be performed on what objects by which subjects. Negative statements define what actions are not allowed.

Negative authorisations allow to efficiently prevent a particular subject from accessing a specific system or resource, as required. Additionally, such statements are often used to impose additional constraints on access control policies and to support various access control models. These policies allow to define such constructs as *conflict of interest* or *separation of duty* [149]. For example a positive authorisation may state that a particular user is able to access an online expense system as well as the online payment authorisation system. A negative authorisation may further state that this user cannot access those two systems at the same time (in order to prevent an expense being submitted and further authorised by the same user).

Positive and negative statements may be defined in a way that a conflict arises. For example, a positive policy grants access to a specific object while the negative policy denies access to this object. In such cases, policy conflict resolution mechanisms can be used. Despite the fact that such conflicts are inevitable in large and distributed computing environments such as the open Web, this thesis does not focus on them. The solution presented in this thesis only supports positive statements.

2.4 Delegated Authorisation

Resources and services in distributed environments, such as those based on the SOA paradigm, and with pre-established trust relationships between various components often do not handle authorisation by themselves but offload this functionality to specialised components. Such externalised authorisation is not a new concept but has been already discussed in various forms in [231; 223; 254]. In [300], authors presented a generalised access control policy language and a contracting protocol for offloading access control from end servers to specialised authorisation servers. In their model, services are required to enforce authorisation but delegate the decision making process to specialised components that are better-suited for this task.

In case of the open Web where applications reside in distinct Web domains under the control of different authorities, such delegation exists only in limited form. Importantly, it does not address the shortcomings that were presented in the previous chapter. The solution presented in this thesis allows applications to establish dynamic bi-directional trust relationships between components that had no prior knowledge of each other. These relationships are established based on preferences of individual users concerned with protecting their data. Additionally, this solution builds on specific features of delegated authorisation that exists in SOA-based computing environments within single administrative. These features are used to allow for Internet-scale access control.

Externalised policy-based authorisation mechanisms rely on four main components [97]:

Policy Enforcement Point (PEP) The PEP component is responsible for enforcing access control policies. It creates a barrier around the resource that it protects and mediates all accesses to this resource. It enforces decisions that are made by other components.

Policy Decision Point (PDP) Evaluates access request decision queries issued by enforcement points. PDP has access to the set of policies and evaluates access requests against applicable policies.

Policy Administration Point (PAP) The PAP components provide administrators or individual users the ability to manage policies in the authorisation system. These components usually have user interfaces or provide high-level tools that support policy composition and management.

Policy Information Point (PIP) PIPs are used to provide information that can be used during evaluation of access requests against applicable access control policies. These components may gather attributes related to subjects, objects and the environment in which access requests are performed.

The proposals presented in this thesis leverage the aforementioned concepts.

There are numerous advantages of externalising security, and access control in particular, and providing these functions in a modular-type architecture for other services and applications to use. Individual applications or even entire systems can be composed from business services and services that address various nonfunctional requirements of the applications. In such applications, business logic is well separated from authorisation which is an orthogonal functionality. Authorisation can be managed independently and plugged easily into more complex distributed applications or systems without the necessity of prior knowledge about relationships between these applications [198]. On the Web, such authorisation can be potentially applied to a distributed set of resources and services.

In enterprise scenarios, security policies can be written independently and these policies do not need to be defined and coded at the same time and in the same package as business services. This facilitates audits and checks of security policies for the purpose of correctness, governance and compliance. As business services in SOA can be used to compose arbitrarily complex applications, such checks and audits are of significant help in anticipating problems before they occur. Changes to security policies can also be easily introduced in order for the computing system to comply with changing security or compliance requirements. This can be done without the need of modifying the logic of business services [198].

In case of applications managed by individual users, externalised access control policies can be applied to potentially distributed resources and services. Therefore, a user could use a single and centralised system in order to apply security to Web data that this user owns or controls. This thesis focuses solely on such user-centric and user-managed cases in the Web environment.

When authorisation is delegated to specialised components then potentially more accurate authorisation decisions can be made [300]. Authorisation services can have a detailed and holistic view of the computing environment and may introduce useful information into the policy evaluation process. Because authorisation services provide their functionality for a group of services, typically the entire computing system, administrative domain, or an ecosystem, a consolidated view of the security that is applied within such a system is also possible. This may support better manageability because standard management tools can be developed for the entire system. Additionally, authorisation services easily contribute to uniformity of accounting and auditing functions and this is discussed in more detail in [300; 158]. The aforementioned characteristics of delegated authorisation match precisely one of the requirements as defined in the previous chapter (see requirement R4). This thesis discusses proposals that rely on such delegated authorisation.

2.4.1 Interactions

Interactions between the decision (PDP) and enforcement (PEP) points can be based on one of the three proposed main authorisation decision query sequences. These sequences have been discussed in existing authorisation frameworks such as the *Generic AAA Architecture* presented in [168], *Authorization Framework* presented in [293], with requirements in [182] and examples discussed in [292], the *Access Control Framework* described in the ISO/IEC 10181-3:1996 recommendation [250] or the *Conceptual Grid Authorization Framework and Classification* proposed in [234]. These decision query sequence models are:

1. Agent model;
2. Pull model;
3. Push model.

The *access control decision agent model* is a proxy-based approach where a specialised component sits in front of an exposed service (resource provider) and mediates all access requests to this service. It is the simplest form of the previously mentioned *reference monitor*, which was first introduced in [224]. It is also the most commonly implemented model in existing Web applications. In this model, the PEP and PDP are collocated together at the service side. The service can only communicate with the agent and does not accept access requests from any other sources (i.e. all access requests have to be mediated through this agent). In the *access control decision pull model*, the access request is made directly to the service, which is responsible for sending it to a decision component, such as PDP, of the authorisation system. When such access request is successfully validated against applicable policies then the client is given access to the service. In the *access control decision push model*, it is the client that communicates with a decision component (PDP) and obtains a capability for a specific service or a resource. The decision component decides if a capability (representing permissions of the client for a particular resource or a service) should be issued to the client using applicable access control policies. This capability is later checked by the enforcement point. In case of push and pull models, the decision component is separated from the client as well as the service.

Importantly, all three authorisation decision query sequences can be combined with either push or pull credential sequence models. In the *push credential sequence model*, it is the client that is concerned with obtaining credentials before the access control process (i.e. before these credentials are used to evaluate permissions of this client for a service or a resource). In the *pull credential sequence model*, it is the enforcement point that obtains the credentials and then uses them during the access control process (see [160]).

The next sections of this thesis discuss various access control decision query models combined with the push credential sequence model. In the provided examples, the client first obtains the credentials and only then an access control decision process takes place, either using agent, pull or push model. The service that issues credentials is called the *Credential Service*, the service that issues capabilities is called the *Capability Service*, and the service that issues access control decisions is called the *Policy Service*. The capability and the policy service are concerned with checking access requests against applicable access control policies. The objective of the provided examples is to present how access control is performed and not how clients are authenticated or how their credentials are validated.

The agent model constitutes a decentralised approach to access control policy management. Policies need to be expressed, managed and enforced in each distributed agent and these agents are located at the perimeter of every domain where services are enforced. A single administrative domain may have multiple subdomains where services are located and may require multiple agents to control access to those services. As depicted in Figure 2.4, a client makes a request to a resource (*step 1*) and this request is evaluated by the policy service against applicable access control policies. This policy evaluation is performed locally within the agent.

In the provided example, policy evaluation is done based on supplied credentials. The client issues a request for credentials (*step I*) and such credentials are provided to the client after successful verification (*II*). These credentials are used in access requests to resources, which constitutes the credential push model. Importantly, the agent (proxy) has an established trust relationship with the Credential Service and therefore it can trust the credentials that are issued by that service.

The access control decision making service (Policy Service) is an integral part of the agent that mediates access to resources. Access to a resource is granted or not (*2*) based on the decision issued by this service. The agent model could be observed in the scenario presented in Section 1.1.1 and this model is considered ill suited for the open Web. The PEP and the Policy Service are collocated and therefore these two components have an assumed trust relationship established between themselves.

In case of access control decision push and pull models, policies can be managed independently from the services. These policies can then be applied to various services located in different domains. Therefore, only these two models fit well into SOA and the open Web. Both solutions provide means of centralising authorisation information in specialised servers [184]. These two particular models have been used during design of the proposals that are discussed in this thesis.

The push and the pull model of authorisation decision query can be presented by comparing two distinct authorisation mechanisms:

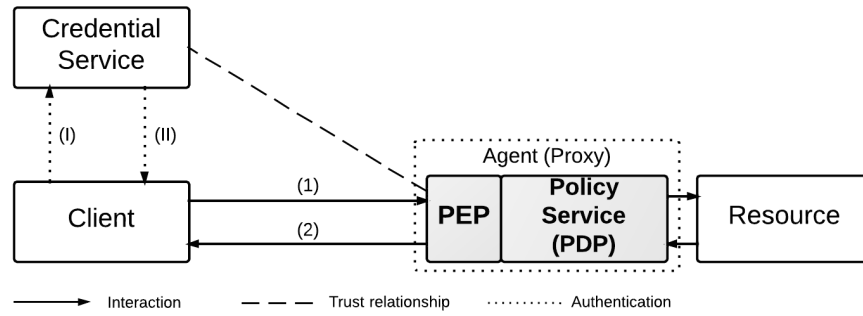


Figure 2.4: Agent model approach to access control [219].

1. Capability-issuing mechanism;
2. Decision-issuing mechanism.

The *capability-issuing mechanism* uses the access control decision push model. A client first obtains appropriate capability to access a resource. Such capability represents the permissions that the client has for a particular resource/service. This capability is then passed along with the access request to the resource. Capabilities can be attached to the access request as an assertion, which upon verification by the service, results in access being granted or not. Kerberos [255] is an example of such system and it is discussed in Section 3.4.

The pull model can be well presented by discussing the *decision-issuing access control mechanism*. In this mechanism, a client is only concerned with invoking the business service. Such service is then responsible for communicating with the appropriate decision component to determine whether access should be granted or not. XACML [97], along with SAML [102], is an example of a decision-issuing mechanism. XACML and SAML are discussed in Section 3.2 and Section 3.3 respectively.

The capability-issuing and decision-issuing approaches to access control are similar in terms of the general structure and syntax. However, these two approaches differ in terms of their execution semantics. They have different trust relationships and interactions between components of the security architecture [219]. These differences are discussed in more details in Section 2.5.1 and Section 2.5.2.

2.5 Access Control Mechanisms

The next two sections present the push and pull model of authorisation decision query by presenting two distinct policy-based access control mechanisms. The *capability-issuing mechanism* uses the access control decision push model while the *decision-issuing mechanism* uses the pull

model. Both access control mechanisms use policies for their authorisation decision making process. The following sections discuss the differences between these two mechanisms.

2.5.1 Capability-Issuing Security Architecture

A capability-issuing authorisation architecture features a trusted capability service [219] that can be used by clients of business services. This Capability Service makes authorisation decisions and can be viewed as Policy Decision Point (PDP). Referring to Figure 2.5, a client issues a capability request (*step 1*) which is evaluated by a Capability Service against applicable policies. A Capability Service replies with a capability response (*2*), acting additionally as a Security Token Service (STS). This response typically includes signed assertions that contain information about the action (or actions) that can be performed by the subject on an object. These signed assertions are collectively referred to as *capabilities*. A response from a Capability Service may also pose additional constraints on the permissions that the client has. An example of such constraint may be the period of time during which a particular permission is valid.

A client (subject), which requested capabilities, includes them when making requests to a business service (*3*). Optionally, a client can include its credentials as well. The capability is extracted on the business service side and it is validated for its integrity and authenticity. Only then the enforcement point checks whether the capability is sufficient for access to be granted (*4*).

In the example visualised in Figure 2.5, policy evaluation is done by the Capability Service based on supplied credentials. The client issues a request for credentials to the Credential Service (*step I*) and such credentials are provided to the client after successful verification (*II*). These credentials are used in the request that the client makes to the Capability Service.

Importantly, there are a number of trust relationships between different components of the capability-issuing architecture (see Figure 2.5). The Capability Service, before issuing the capability to the client, has to trust the provided credentials. Therefore, this service has an established trust relationship with the Credential Service. Secondly, there is a trust relationship between the enforcement point and the Credential Service as well as the Capability Service. Firstly, the enforcement point can validate if access requests come from trusted clients (i.e. the PEP can verify that calls to services are authenticated and that credentials can be trusted). Secondly, the enforcement point can validate if the capability that is sent by the client can be trusted. The trust relationship between the PEP and Credential Service is optional when access requests made by client applications to this PEP do not contain any credentials. The enforcement point of a resource provider needs to have access to trusted public key certificates of credential and capability services and is able to technically assert a trust relationship with

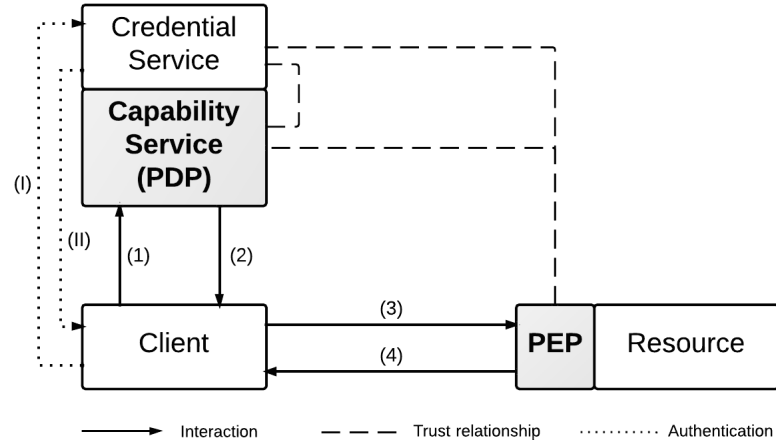


Figure 2.5: Capability-issuing approach to authorisation in computing environments [219].

those services. Another approach is where a secret key is shared between the enforcement point and the capability and credential services. The same principles apply to the trust relationship between the Capability Service and the Credential Service.

Capabilities can be encoded by the Capability Service using different formats depending on the actual deployment and the underlying protocol. In case of Web Services in enterprise settings, capabilities can be encoded as SAML assertions [102] or even X.509 certificates [206]. In the modern Web, the use of access tokens in various forms (e.g. JSON-encoded tokens [39]) is more popular. In all cases, assertions are included in requests to resources. For example, an assertion can be included in a header of a SOAP message that is sent by a client. An assertion can be also included in a header of HTTP request to a RESTful Web Service. More lightweight, but less secure, proposals allow the use of assertions in HTTP queries.

In the capability-issuing approach, the policy decision making process may be distributed within the computing environment. Even though the capability service may assert that a client can access a resource, the resource provider may still make the final access control decision. This allows using the capability service to prescreen clients and issue capabilities based on general information. The resource providers may impose their own restrictions on access requests. Similar approach has been included in the Street Identity protocol discussed in Section 3.8.

There are two well known examples of a capability-based access control systems. Those are the Community Authorization Service (CAS) [268], which provides security for Globus, and Kerberos [255] that is widely used within enterprises. Both solutions differ with respect to the format of the capabilities that are issued and the granularity of capability-enriched access requests. The CAS system uses X.509 proxy certificates for capability encoding while Kerberos uses its own format for this purpose. On the Web, OAuth 1.0 and OAuth 2.0 protocols have

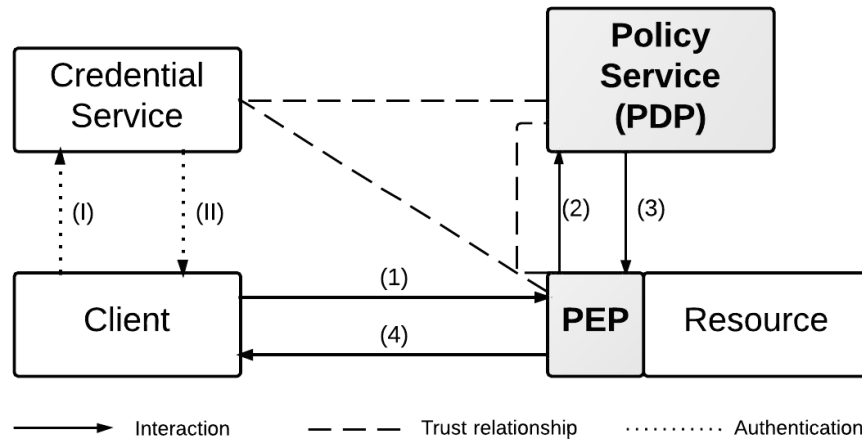


Figure 2.6: Decision-issuing approach to authorisation in computing environments [219].

been proposed and these proposals are discussed in Sections 3.5 and 3.6 respectively.

2.5.2 Decision-Issuing Security Architecture

In decision-issuing authorisation architectures, a client is not required to obtain any capabilities or to pass them to a resource that this client tries to access. Access requests are made as usual by client applications and it is up to the security mechanism on the service side to obtain a decision whether those requests should succeed or not. A decision-issuing security architecture is depicted in Figure 2.6.

Referring to Figure 2.6, when the enforcement point intercepts an access request (*step 1*), it describes it by creating an authorisation decision query that is sent to a decision point (*2*). This authorisation query contains information about the subject of a request, be it a user or another service, the target and action which has been requested (e.g. reading or writing). PDPs can request additional information about the environment in which the access request is being performed. This may include time of access request or possible history of previous access requests made by this client [171]. PDP evaluates access requests and decides whether access should be granted or not. Authorisation decision response is returned to the enforcement point (*3*) which adheres to this decisions and optionally fulfils additional obligations. PEP then grants or denies access to a resource (*4*).

Single PDPs can be used for each administrative domain, while PEPs need to be located in every place where access control should be enforced. PDPs have engines to determine whether access to the resource should be granted or not. A PDP evaluates an authorisation query against a policy or a set of policies that are retrieved from a policy repository handled by PAP. PDPs additionally obtain information from PIP components and this information can be used during

the decision making process.

In the example visualised in Figure 2.6, policy evaluation is based on supplied credentials - i.e. the client issues a request for credentials (*step I*) and such credentials are issued to the client after successful verification (*II*). These credentials are used by the client when making access requests to the resource. The PEP component extracts these credentials and provides them to the Policy Service. The PEP does not have to be concerned with the trustworthiness of the credentials supplied by the client.

Importantly, there are various trust relationships between different components of the decision-issuing architecture. Firstly, the Policy Service, before issuing an access control decision, has to trust the provided credentials. Therefore, this service has an established trust relationship with the Credential Service. The PEP component has to trust decisions that are issued by the Policy Service and has an established trust relationship with this service. This trust relationship is mutual as the PDP should only issue decisions to trusted PEP components.

Optionally, the PEP component may want to trust credentials that are passed by the client in the access request to the resource. Therefore, this component may also have an established trust relationship with the Credential Service. In such setting, the PEP can validate if access requests come from trusted clients (i.e. the PEP can verify that calls to services are authenticated and that credentials can be trusted). This trust relationship, however, is optional when access control decision making is fully delegated to the Policy Service.

2.6 Access Control Challenges

Building secure and dependable authorisation systems for multi-domain computing environments poses numerous challenges. Such environments have significantly different requirements for access control than those in closed systems, as discussed in the previous chapter. In particular, authorisation must not only follow requirements that are common for SOA and the Web such as modularity, extensibility or re-usability [140]. Such access control needs to span separate and autonomous domains of administration, scale to large base of users, resources and services, and should be efficient enough to handle even fine-grained interactions between distributed components [211; 240].

The next sections discuss common access control challenges and requirements, which are orthogonal to the previously presented requirements **R1 - R4** introduced in Chapter 1. These challenges were discussed in [240] and are as following:

1. Policy challenges:
 - (a) Heterogeneity and distribution of subjects and objects;

- (b) Context and content-based access to resources;
- (c) Policy heterogeneity management;
- (d) Policy conflict resolution;

2. Architecture challenges:

- (a) Interoperability between access control components;
- (b) Location of policy decision points;
- (c) Management of access control systems;
- (d) Communication performance;
- (e) Autonomy of administration domains;
- (f) Access control delegation;
- (g) Attribute aggregation;
- (h) Authorisation credential issuing and validation;
- (i) Assignment of arbitrary authorisation attributes;
- (j) Security of access control systems.

The relationships between the previously identified requirements and the challenges discussed here are shown in Table 2.1.

2.6.1 Policy Challenges

Policies that define access control rules need to address a wide spectrum of subjects and objects that the authorisation system aims to protect [240]. As far as subjects are concerned, access control policies must be able to scale to a large user base and should be able to address the heterogeneity of subject credentials. Such heterogeneity is inevitable on the open Web.

Importantly, access control should be flexible and should allow decisions to be made based on attributes of subjects and not only on their identities as those may not be known a priori in dynamic distributed environments. As far as objects are concerned, authorisation policies should be applicable to their wide spectrum. Such policies should allow newly created resources to be easily protected with the existing authorisation system.

Moreover, the access control system should provide means of making decisions based on multiple policies. Such policies in multi-domain environments are commonly defined with different syntax and semantics. In such cases, interoperability at the level of access control policies is a necessity. Moreover, policy conflict resolution protocols must exist to support decision making

Table 2.1: Relationships between access control challenges and requirements for a user-managed access control proposal.

	R1	R2	R3	R4
Heterogeneity and distribution of subjects and objects	✓		✓	✓
Context and content-based access to resources	✓		✓	✓
Policy heterogeneity management	✓	✓	✓	✓
Policy conflict resolution		✓	✓	✓
Interoperability between access control components	✓			✓
Location of policy decision points	✓		✓	✓
Management of access control systems	✓	✓	✓	✓
Communication performance	✓			✓
Autonomy of administration domains	✓			✓
Access control delegation	✓		✓	✓
Security of access control systems	✓			
Attribute aggregation		✓	✓	✓
Authorisation credential issuing and validation	✓			
Assignment of arbitrary authorisation attributes		✓		

process when applicable policies come from separate administrative domains because these policies may contain contradicting rules. Centralised authorisation systems partially alleviate this problem, which is discussed in this section.

2.6.1.1 Heterogeneity and Distribution of Subjects and Objects

The access control infrastructure should be able to address security of a highly distributed environment of heterogeneous subjects and objects [211]. It must also scale to a large number of objects. These requirements hold particularly for the open Web, which is the largest existing computing environment.

On the Web, it is necessary to be able to specify different access control rules for different methods which are normally invoked by issuing requests directed to a single URL as in SOAP-based Web services. It is necessary to provide authorisation based on the content of messages that those services exchange. The Web Services profile for XACML [107] defines policy assertions that can be used for specifying authorisation and privacy requirements. Such assertions may constitute policies that are specified at the Web Service side using the WS-Policy framework [232].

In case of RESTful Web Services, which have been proposed as a more lightweight approach to SOA-type integration of computing systems, similar profiles to those available for SOAP-

based Web Services are not necessary. Following constraints imposed by the Representational State Transfer (REST) architectural style, Web Services are accessed using different URLs and HTTP methods and it is easier to control access to these services. The UMA proposal that is discussed in this thesis allows for flexibility in specifying resources as well as actions on these resources, and these can be defined during runtime.

As far as heterogeneity and distribution of subjects are concerned, the authorisation system needs to take into account that subjects' credentials can be issued by IDPs from separate administrative domains. There are various ways of ensuring that different IDPs are capable of issuing credentials that can be trusted by authorisation components. Most of these approaches rely on the Public Key Infrastructure (PKI) [134] which constitutes a fundamental block of building trust between collaborating parties. These approaches of trust between the authorisation system and user credentials can be:

1. Identity-based;
2. Capability-based;
3. Trust-negotiation based.

In *identity-based access control* systems, it is the user that presents its own identity credentials and the authorisation system may trust the party (Identity Provider) that assures this identity. The authorisation system may simply contact the IDP and ask for all the information, collectively referred to as profile, that it requires to make an access control decision [144]. In this approach, the service gets to know the identifier of the user and attributes (claims) associated with this identifier.

In multi-domain computing environments such as the open Web, the user base is very large and defining access control rules based on specific identities may not always be efficient (importantly, this differs depending on the exact use cases - e.g. sharing across a group of friends on the Web can easily leverage user-defined contact lists, which are based on identifiers). Therefore, clients that issue access requests to services only provide their identifier so that it can be used to obtain attributes, which are used to make access control decisions. Such attributes may be in form of roles that the user is entitled to activate during a particular access request, or contain personal information that is used for policy evaluation (e.g. verified date of birth information is used during the decision making process).

In case of *capability-based access control* systems, the user does not use their identity but obtains the required capabilities from a capability service and sends them to the enforcement point. Such capabilities are later used during policy evaluation. This approach is widely used

in enterprise settings, e.g. in the CARDEA authorisation system as presented in [235], and it is being introduced to the Web with such protocols as OAuth.

In capability-based systems, access requests are evaluated dynamically according to a set of relevant characteristics of the client rather than considering specific local identities. This reduces reliance on local identities to define authorisations for each potential user [240]. In capability-based systems, it is possible to define the set of required attributes that the client needs to present to the service. As in identity-based systems, those can be in form of roles, which the user activates to perform a particular access request on a service. Similarly, these can be verified claims such as the previously mentioned date of birth information. However, in capability-based systems, these can be also permissions that a client must have for a particular service or a resource.

In case of highly dynamic multi-domain computing environments, user management may be very complex. Neither identity- nor capability-based approaches of determining one another's trustworthiness may provide the required functionality in terms of their ease of management, user provisioning or scalability. In such situations, it may be necessary to provide mechanisms where trust could be established without the assumption of familiarity of collaborating parties. Collaborating parties may use trust negotiation to assure the enforcement and decision points that they indeed should be granted access to a particular resource. In this process, the client and the resource provider (or the decision point) conduct a bilateral and iterative exchange of policy information and credentials/claims to incrementally establish trust between themselves [227]. A more detailed explanation of trust negotiation is discussed in various publications including [299], [226] and [227]. The last one presents an approach where specialised components, called *Trust Servers*, handle trust negotiation processes and determine which users are authorised to access resources within their protection domains.

2.6.1.2 Context and Content-Based Access to Resources

Controlling access to resources in a multi-domain computing environment involves defining complex access control policies. It should be possible to specify restrictions on access requests based on the context of such requests. Moreover, it should be possible to declare arbitrarily complex conditions under which an access should be granted or denied. Information that can be useful for specifying authorisation policies may include users' administrative domains, time of access, or an environment state [211].

As depicted in Figure 3.1 in Section 3.2, the authorisation decision query sequence specified by the XACML model involves retrieval of attributes through the PIP component. Those attributes may refer to the subject, resource, action and environment. PIP components may

provide those attributes, particularly those that refer to subjects and resources, and may calculate attributes for actions and environments during the access request. PIP components may provide attributes from a number of IDPs depending on the configuration or on the actual access request that comes from the PDP. The UMA proposal, discussed in Chapter 5, envisions that the context may be used in the access control decision making process. The UMA implementation, which is discussed in Chapter 7, does not yet support this feature.

Access control rules should also be able to address the need of a content-based access. Such requirement, as presented in [211] in the context of XML-based multi-domain environments, should allow resource owners or custodians to specify restrictions depending on information contained within the resource. Similarly, Sun et al. [286] recognises such need in the open Web, where access control should allow a content owner to protect a photo in an album, an event in a calendar, or even a paragraph within a blog. The UMA proposal allows applications to define resources at any level of granularity. For example, an application can protect an entire blog post or a single paragraph in that blog post.

In dynamic Web Services, it is virtually impossible to predict the content of the resource which is requested by the client because such content can vary. This makes content-based access control more challenging. Certain access control policy languages, such as XACML, incorporate the concept of obligations, which could be used to provide the required content-based authorisation. As discussed in Section 3.2, obligations are actions that must be executed before, with or after an authorisation decision. XACML does not constrain obligations and allows them to be implementation specific. Therefore, such obligations could be used to provide the required level of content-based access control.

For example, in the context of Web Services, when a resource is requested then the hosting application can be informed that access can be granted but an advanced check on the content of the resource should be performed. These requirements would be provided in form of an obligation. The required advanced check would determine whether the client can access the resource. The access control proposal that is discussed in Chapter 5 supports applications hosting data to be able to have the final decision whether access should be provisioned or not, thus partially providing solution to content-based authorisation.

2.6.1.3 Policy Heterogeneity Management

When distinct administrative domains create a federated environment, most typically each domain has their own security policies (called *local policies*). On the Web, in particular, local policies exist in every single system, which is used by the individual user. As discussed in [211], integration of local policies that are distributed entails various challenges including reconciliation

of semantic differences between these local policies, secure interoperability, containment of risk propagation and policy management.

There are various potential ways of approaching those challenges as discussed in [211] and [212]. Management of policies, including access control policies, in multi-domain computing environments could be addressed by employing one of the following approaches:

1. Policy mapping;
2. Standardised policy language.

In the first approach, a global authorisation policy is defined for the entire computing environment and its rules are then applied to each individual domain. For example, a global role can be mapped to local roles within different domains. Local policies do not have to be in the same format or language as the global policy as long as the mapping can be done correctly. However, the use of a standard language, such as XML, simplifies policy management because XML allows for uniform representation, interchange, sharing, and dissemination of information in heterogeneous environments. Authors in [211] recognise the challenges of managing semantic heterogeneity and integration of multiple different policies in an XML-integrated multi-domain computing environment.

The other approach is to enforce usage of a standard policy language that would be used consistently throughout the entire security system. With a standard policy language, access control rules from different domains would have a uniform representation (e.g. in XML).

In multi domain computing environments, organisations are moving towards standardising their authorisation policies and the latter approach is more favourable. This is discussed on the example of XACML in Section 3.2. In the open Web, however, relying on a single standard policy language in distributed applications is unrealistic because of the number of different Web applications. Therefore, this thesis discusses a solution which centralises access management for distributed resources but still allows for high-level of flexibility to be achieved by different Web applications. In the discussed approach, policy mapping, uniform policy representation or standard language are not required to address the challenge of policy heterogeneity. Instead, the user is able to define their own policy using their preferred policy management tools and in the language of their choice using a central component. Such policy can be later applied to distributed online resources.

2.6.1.4 Policy Conflict Resolution

Policy decision points evaluate access requests against all applicable policy sets, policies or rules, which are retrieved from designated administration points (Section 2.4). In distributed

computing systems, each domain typically has its own set of PAPs where different authorities or individual users define access control policies (refer to Figures 2.2 and 2.3). The same applies to the Web, where individual systems or entire Web ecosystems have their own PDPs as well as PAPs. PDPs and PAPs allow to centralise policies that can be later applied to resources that are distributed.

It is typical that multiple distinct authorities are supported by a single authorisation system. Decision components of such system may wish to retrieve policies that reside in various policy repositories. Such solution has been adopted by the PRIMA authorisation system where users as well as administrators from different domains are able to delegate authorisation for resources for which they are authoritative to other authorities [235].

As policies, which are evaluated during the access control decision making process, may come from different authorities then policy conflicts may arise. This is due to omissions, errors or conflicting requirements of parties specifying those policies [236]. When multiple policies or rules apply to the same access request then it is possible that inconsistencies may occur. This can happen when two or more policies apply to the same subject, operation and to the same resource and their rules are contradicting. An example is when one policy states that a particular access request is valid while another policy forbids such access.

Policy inconsistencies may result in illegal accesses to resources or in legal accesses being prevented. Therefore, ensuring that policy conflicts are resolved is an important issue. Certain conflicts can be resolved before policies are deployed within the computing environment (i.e. before those policies are sent to PAPs). This process, called the static conflict resolution, is based on static analysis of policies and applies only to modality conflicts [236]. Such analysis enumerates all $\{subject, action, object\}$ tuples which have a different set of applicable policies. If there are two or more policies applicable to a tuple then there is a potential conflict that may need to be resolved. These policies must be later checked to see whether there is an actual conflict, i.e., a positive and negative policy with the same subjects, targets and actions [236; 214].

As stated in [97; 229], conflict resolution in XACML is addressed with the use of rule and policy combining algorithms, which are used for making authorisation decisions based on policy sets and multiple rules. XACML defines following algorithms for this purpose: *deny overrides*, *permit overrides*, *first applicable* and *only one applicable*. When an XACML-compliant decision point finds two or more policies or two or more rules within a single policy with contradicting semantics then it uses one of the mentioned algorithms to make its access control decision.

Static analysis cannot point out all conflicts in access control policies. Some conflicts are application specific and are usually visible only at runtime once all policies are deployed within

the system. An example of such conflicts is when rules defined in policies do not address the required principle of *Separation of Duty (SoD)* [149]. Such principle may be imposed by an organisation that collaborates in a multi-domain computing environment. One of the proposed solutions to the problem of application specific conflicts are meta-policies that contain constraints on other access control policies [236].

Considering access control mechanisms discussed in Section 2.4 and the environments depicted in Figures 2.2 and 2.3, meta-policies can be placed both within each domain or can be applied to all domains. In the first case, constraints usually refer to the SoD principle where the same client should not be able to access certain resources at the same time. In case meta-policies are used for the entire multi-domain computing environment then policies may address conflict-of-interest type issues according to the model proposed by Brewer and Nash [150]. When a certain collaborating party decides to access resources from one domain then this party is prevented from accessing any resources from a different domain within this computing environment.

2.6.2 Architectural Challenges

The access control architecture in distributed environments needs to take into account arbitrarily complex integration scenarios between organisations, ecosystems or applications that form the environment. It needs to address the heterogeneity of components that comprise such architecture and has to provide means to ensure interoperability between them. Components should be able to exchange information meaningfully when interacting during access control related tasks.

The authorisation architecture should additionally be secured in a similar way to resources within the computing environment. Communication between components should be dependable with confidentiality, authenticity and integrity of messages being preserved. Such architecture should be capable of scaling along with the environment itself while preserving efficiency of communication between its components. With the ever growing number of applications and data generated by users on the Web, for example, such scalability characteristics are of pivotal importance. Challenges in access control architectures are presented in more details in further sections.

2.6.2.1 Interoperability Between Access Control Components

Separate and autonomous administrative (Web) domains need to cooperate in a distributed authorisation system. Such a system needs to maintain a consistent authorisation strategy and each domain should have at least some knowledge of its potential collaborators throughout the entire lifecycle of the multi-domain computing environment. Authorisation decisions that span administrative domains require that components in every domain are capable of correctly producing,

accepting and interpreting authorisation information from a group of potentially heterogeneous peers. A common agreement protocol, syntax and semantics of every piece of information that is exchanged between components of the authorisation system is a necessity [235]. This includes interoperability at the level of language that is used for specifying permissions and at the level of protocols that are used for communication between various components of the system.

Achieving interoperability at the level of policy specification language has been the subject of much research and numerous languages exist for specifying access control rules [97; 106; 148; 199; 147]. The XML technology has emerged as one of the most promising approaches for those languages [211]. XML allows uniform representation, interchange, sharing and dissemination of information between heterogeneous systems within an environment. XML constitutes the base for such access control policy languages as XACML (see Section 3.2). XML is also used by XACML to exchange messages between components of the XACML architecture³. The open Web has adopted a more lightweight JSON-based approach, which is mostly used for the purpose of exchanging authorisation data between different systems (and not necessarily for specifying access control policies that can be coded using XML). In the UMA proposal, components communicate with each other using JSON-based messages, while policies can be defined using a totally different technology (e.g. the presented UMA implementation stores policies within a relational datastore).

Standardising protocols that are used for communication between distributed components of the authorisation system is a necessity. Such standard is presented in this thesis. As discussed in Section 3.2, information flow between enforcement, decision, information and administration points of the access control system requires multiple messages to be exchanged. Components of the authorisation system must agree on the syntax and semantics of information that they wish to exchange in order to interoperate. The request/response protocol proposed in the XACML standard aims to achieve interoperability but is ill-suited to the open Web. Chapter 5 discusses a standard that fits precisely to the user-driven Web environment.

Apart from the syntax and semantics of information exchanged between various components, interfaces of those components should be standardised as well. This also applies to components which expose their functionality as SOAP-based or RESTful interfaces. In both cases those interfaces must be well-defined and other components of the access control system must be able to interact with them. In case of XACML-compliant components of the authorisation architecture, interoperability of components and their interface definitions have been discussed in [108] and [112]. The UMA proposal, as discussed in Chapter 5, defines these interfaces for interoperability purposes. Moreover, applications can discover the location of these interfaces at

³A standard that defines JSON-formatted request and response messages between components in the XACML architecture has been proposed in [125].

runtime.

2.6.2.2 Location of Policy Decision Points

Once an access request is made to a resource, the enforcement point needs to contact the decision point to determine whether access should be granted or not. The enforcement point needs to know which decision point should be used. In distributed systems with a limited number of components of the authorisation infrastructure, the relationship between PEP and PDP components can be static (e.g. configured by the IT administrator). When the PEP is initialised, it simply checks whether a pre-configured PDP is available and a communication channel is created with this point [300].

Decision and enforcement points have to form a mutual trust relationship between each other in order to interact. Both components must undergo a mutual authentication, e.g. based on a shared secret or using PKI. This mutual authentication is necessary because of two reasons. Firstly, enforcement points need to be sure that authorisation decisions come from trusted decision points. This guarantees that enforcement actions reflect applied security policies. Secondly, decision points should only reveal decisions on authentic access request decision queries. Otherwise, these systems may potentially leak information about access control policies in place.

Although static binding between enforcement and decision components in small distributed systems is sufficient, it does not fit into large computing environments spanning multiple separate administrative domains. At first, PEPs may delegate rights to other domains and may not wish to specify exactly which PDPs should be used. Instead, PEPs may rely on individual users to provide information on their preferred PDPs that should be used to protect this user's resources. Moreover, in case of large and dynamically changing distributed systems, a static binding between PEPs and PDPs may not be feasible. In such cases, a discovery mechanism needs to be employed.

This thesis discusses a dynamic approach that allows enforcement points to form relationships with decision points in an ad-hoc basis. These relationships are created based on preferences of users. Communication channels between enforcement and decision points use SSL/TLS as well as access tokens to meet the requirement of mutual trust relationship.

2.6.2.3 Management of Access Control Systems

Configurations of different enforcement points in an access control system are often managed independently in order to implement the security policy as accurately as possible. In such cases, each enforcement point is configured with their own security policy specific to a resource or a set of resources that this enforcement point protects. Consequently, it is an expensive and

unreliable proposition to modify the security policy and deploy it within the entire environment. Additionally, it is virtually impossible to obtain a consolidated view of the safeguards and security controls that are deployed within the entire enterprise or the entire computing environment, such as the open Web. This may result in a poor understanding of the security rules that are actually enforced.

In enterprise settings, there is an increasing pressure on executives from customers, shareholders and regulators to demonstrate best practice in the protection of the information assets of the enterprise and its customers. Executives are forced to assure that security is provided in a way that follows guidelines from such documents as ISO2007 [95; 96] and is in line with necessary legislation requirements such as the UK Data Protection Act (DPA) [288]. Similarly, on the open Web, users are increasingly concerned with security and privacy of their growing number of resources. These users may want to verify how security is applied to their resources but also how such resources are accessed as well.

Access control systems need a way of providing a consolidated view of the authorisation policies that are enforced within a computing environment. This can be achieved by centralising PAP and PDP components. Policies can be composed independently of the services for which these policies are used. In enterprise settings, those policies can be defined to meet compliance or governance requirements. In the context of the open Web, such policy centralisation allows individuals to easily define their security requirements for online data. Moreover, centralising policies guarantees that those policies are applied consistently among a set of distributed resources. Policy centralisation also facilitates audit of security rules.

In enterprise settings, for example, a corporate LDAP can be used for storing a single policy or a set of policies that are later pulled by local PDP components. However, a single policy is neither feasible nor desirable on the open Web. In this environment, a more favourable approach is to centralise access control policies. Centralising policies and keeping them separate from the actual services or resources provides benefits to individual users, which is discussed in this thesis.

Moreover, as discussed in [97], policy management in access control systems involves many different steps. These steps include writing, reviewing, testing, approving, issuing, combining, analysing, modifying, withdrawing, retrieving and enforcing authorisation policies. Providing means of securing all those steps should be considered mandatory in an access control solution.

The proposals that are discussed in this thesis centralise policy management and policy evaluation. Therefore, they allow for consistent security to be applied across distributed Web applications used by an individual user. These proposals also support audit allowing the user to quickly verify how security is applied to various services or resources on the Web.

2.6.2.4 Communication Performance

Authorisation decision is often based on information from different distributed components. This situation occurs when enforcement points reside in different domains than decision points and the decisions points additionally collect information from a distributed set of information points. In such scenarios, it is necessary to ensure that communication between components of the authorisation system is efficient in terms of the number of messages that are sent between components and the size of those messages.

The protocol used for offloading authorisation from business services should be based on the exchange of a possibly small number of messages. In decentralised approaches to authorisation, access request received by the enforcement point may need to be encapsulated in some form of access request decision query and sent for evaluation. In such case, minimising the number of interactions between components can be achieved using caching, as proposed in [300]. PEPs may cache decisions made by PDPs. Additionally, PDPs may cache policies that they would normally retrieve from PAPs.

Caching can significantly reduce the number of messages that are exchanged between access control components but has certain drawbacks. At first, information stored in the cache memory may not be up-to-date which may result in false positive or false negative access control decisions. This reduces the flexibility of revoking old access control rules or introducing new ones. This problem can be minimised by introducing time constraints on validity of locally cached copies by different components of the system. Moreover, components may only decide to cache information which does not change frequently but is used more often than other information (e.g. only a subset of infrequently changed but commonly used policies can be cached).

Importantly, caching can provide significant benefit when the cached information refers to commonly accessed resources. For example, a PEP may cache an access control decision that has been made in relation to the entire resource set. This cached decision can then be reused when access requests are made for individual resources from this particular set. Similarly, if a resource changes frequently and is being accessed on a continuous basis then caching of the decision for this particular resource can be desirable.

The proposals that are discussed in this thesis allow the use of caching. Initially, in UMAC this would be based on the existing HTTP caching mechanisms, i.e. **Expires** or **Cache-Control** headers in HTTP response. The refined UMAC proposal as well as UMA uses tokens which would have a specific lifetime of validity.

Additionally, minimising the number of messages in the access control system can be achieved through policy syndication, as proposed in [285]. A global PAP, which is managed centrally, may hold a global security policy. Such policy is then syndicated to more local PAPs residing in

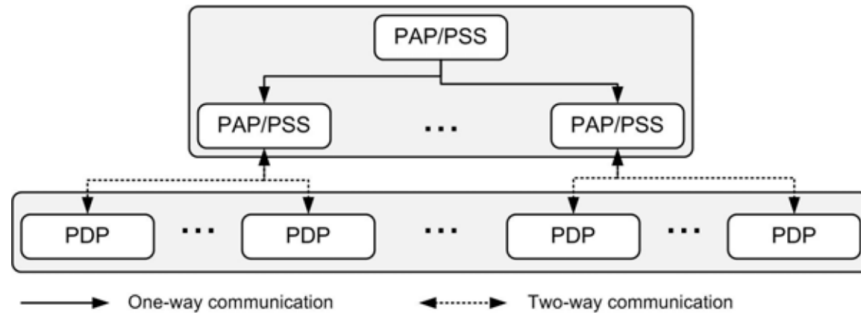


Figure 2.7: A high-level view of the Policy Administration Point / Policy Syndication Server hierarchy.

different administrative domains or in the same domain in which syndication takes places. These local PAP components can incorporate all changes or only those that are in line with constraints imposed by authoritative bodies of those local PAPs. A hierarchy of such PAP interactions can be created as depicted in Figure 2.7.

This thesis does not aim to discuss policy syndication in further details. The presented UMAC and UMA approaches use delegation that allows a policy to reside in a single location and relies on enforcement or decision points to refer to authoritative sources to make meaningful decisions regarding access requests.

Communication between components of the authorisation infrastructure should also aim to be based on lightweight messages that would not affect the overall throughput of the computing environment. Such characteristic is particularly important with regards to authorisation built on Web Services and deployed on Internet-scale. Messages used for carrying access control decision queries or responses can be secured with Web Service-compliant standards and these messages are significantly bigger than those which do not use any security mechanisms [213].

In particular, exchanging security policies between components may constitute a drawback as policies are likely to consist of multiple access control rules. Similarly, communicating access control decisions can affect performance if messages contain vast amount of information. XACML, for example, uses XML to encode access control policies, which results in the size of policies and privilege statements being significant due to the XML encoding overhead and verbosity of the language. The system discussed in this thesis uses a lightweight approach that is well-suited to the Web environment where information can be passed in HTTP headers (in particular, the `WWW-Authenticate` and `Authorization` headers) as well as using JSON-formatted messages (as discussed in [261; 296], this format is significantly more efficient in distributed environments and Web applications in particular).

2.6.2.5 Autonomy of Administration Domains

When resources are shared between different systems residing in distinct domains then access control is delegated in a cross-domain fashion. One domain may decide to accept access control decisions that come from a different domain. Therefore, access control components may form trust relationships that span such multiple domains. Importantly, components that interact with each other should be able to preserve their autonomy in making access control decisions. Each domain will typically introduce their own access control rules that should be enforced for each access request.

Approaching the challenge of preserving autonomy of separate domains can be done with the use of authorities hierarchy. Each level in such hierarchy can be responsible for defining access control rules for a different scope. An example of this approach is discussed in [235] in relation to the PRIMA authorisation system. In this system, multiple entities are authoritative for resources at a different level. For example, a site wide policy can be defined by the site authority and a policy for single resources can be defined by individuals. In addition, there may be authorities that have control over authorisation policies for entire multi-domain computing environment (e.g. Virtual Organisation) or a specific Web ecosystem.

Each level of authorities may impose its own constraints on acceptable access control rules. When such constraints result in conflicts then those can be resolved using some of the proposed policy conflict resolution mechanisms, with examples presented in [235]. The XACML standard provides profiles that could address requirements regarding autonomy of administrative domains. Those profiles extend appropriate schemas to describe administrative requests and delegation policies [109; 113]. A detailed discussion on delegation in authorisation systems is provided in the next section. Importantly, the proposals that are discussed in this thesis support such autonomy. Furthermore, these proposal allow components storing the data to make final decisions regarding access requests.

2.6.2.6 Access Control Delegation

As discussed in the previous section, a collaborating party within a Virtual Organisation may delegate access control decision making process to a different party of the same VO. Such delegation is typically specified in administrative policies that define who is authorised to compose access control rules for resources [275]. A centralised administrative policy is not sufficient for multi-domain computing environments because collaborating parties may not agree upon a single authority to grant and revoke authorisation permissions. In such cases, domains may use cooperative type administrative policies and such policies require all collaborating parties to agree on authorisation rules.

Another approach that is well suited for multi-domain computing environments is a decentralised model of administrative policies. In this model, each domain has its own administrative policy and defines how much of its access control decision making process should be delegated to other domains. When such access is delegated to other domains then those domains may or may not be able to delegate it further. This complicates the authorisation management process because it may be hard to track the actual rights for resources. As discussed in [275], revocation of access control rights is also complex in such cases.

An example usage of delegation is that proposed in the PRIMA authorisation system [235]. This system supports multiple authorities by allowing users as well as administrative personnel to delegate access to resources for which they are authoritative. The resource authorities can use the same mechanisms to grant privileges to other users and to issue policy statements for resources. The XACML standard also provides means of achieving delegation through the use of profiles. The proposals that are discussed in this thesis support delegation. This is shown on the example of a custodian imposing security constraints on distributed Web resources (refer to [243] for more details).

2.6.2.7 Attribute Aggregation

In distributed computing environments, it is often the case that authorisation is based on arbitrary user attributes rather than on user identifiers, as the latter approach does not scale well for large numbers of users. For example, access control to a particular service can be granted to users who are within a specific location, have a particular role assigned or are members of a particular institution. These attributes can come from a single source (IDP) or multiple different sources. The user, when trying to access a service, has to present the necessary third party asserted attributes. The service can then use these attributes in an access control decision making process (either locally or using a remote service such as a PDP).

Existing authorisation systems, which allow for attribute-based access control, support policies that define which attributes must be presented in order to get access to a service or a resource. However, these systems often support only individual authorities for these attributes (i.e. users can present attributes from a single IDP). Even if policies allow to define attributes that should be asserted by different IDPs, there are currently multiple different approaches to aggregating user attributes and presenting them to the resource provider so that these attributes can be used in the access control decision making process. A more thorough explanation of existing attribute aggregation models is presented in [162] and [183].

When designing an access control system, it is necessary to understand various attribute aggregation proposals. Providing a way for users to aggregate their attributes and present

them to resource or service providers is desirable. This is because individual users often have relationships with numerous IDPs and their attributes from these providers could be successfully used in authorisation policies. Challenges related to such aggregation are presented in [162] as well as in [183] and the discussion below is based on these publications.

As discussed in [162] and [183], challenges include aspects of usability, privacy, user consent, protocol, and trust. Firstly, the system should allow the actual aggregation to happen using various means (e.g. via Web browsers or mobile clients) while requiring a minimum amount of interaction from the user. The system should protect the user's privacy through the use of technical controls, which are independent of legal means. Data shared with resource or service providers should be shared only based on the user's consent. Moreover, interactions should be able to span multiple domains and be able to be tunnelled through firewalls using existing ports such as HTTP or HTTPS. These interactions should also extend existing protocols rather than being based on new ones. Furthermore, service providers should be able to require authorities to digitally sign the attributes to have high-level of confidence in the provided attributes. Importantly, each model, as discussed in [183], has its own strengths and weaknesses, which must be considered for a particular access control solution.

2.6.2.8 Authorisation Credential Issuing and Validation

In order for the client to be able to access a resource, this client has to be in possession of some credentials that will allow the resource provider to assess if the client has the necessary rights. As discussed in [160], authorisation credentials may be pushed to the provider by the client or pulled by the provider from a credential issuing service, or a mixture of both. Applying the right model for an access control solution is critical. Therefore, it is desirable to understand the issues that relate to each model.

As discussed in [160] and [161], credentials are issued by a Credential Issuing Service (CIS) and validated using the Credential Validation Service (CVS). There can be at least four different models in which these services are used in conjunction with PEP and PDP components. Firstly, there can be a case where credentials are pulled by the client from CIS and then pushed to the actual resource. In this case, the PEP can either decide to validate these credentials directly at the CVS and only then consult with the PDP whether access should be granted or not. Alternatively, the PEP can offload the task of credential validation to the PDP and it is then up to this PDP to interact with the CVS component for the purpose of credential validation. The PDP then issues an access control decision to the PEP.

Secondly, the client can interact directly with the service provider and credential issuing and validation happens at the service side. The PEP can interact with the CIS and CVS for the

purpose of obtaining credentials and then validating these credentials respectively. The PEP can then use validated credentials at the PDP to obtain the access control decision. Alternatively, the PEP can only pull the credentials from the CIS and then pass these credentials to the PDP component. The PDP uses the CVS for the purpose of credential validation and can reply with an access control decision that should be enforced by the PEP component. Importantly, the CIS and CVS are often colocated and can be a part of the Security Token Service.

The CIS has to have a policy that defines which parties are entitled to receive the issued credentials and such policy is called an attribute release policy. Some services (such as credit card issuers) may only issue credentials to their rightful holders. Other services (such as the Shibboleth [72]) may issue them to trusted service providers. This is determined by the actual attribute release policy [160].

The described models of issuing and validating credentials during access control differ in trust relationships between components as well as in the actual execution semantics. Importantly, access control solutions have to consider where trusted connections can be made or where digitally signed messages can be exchanged.

The UMA solution, as discussed in this thesis, adopts a credential push model where the client pulls credentials from CIS and then interacts directly with the service, including a capability in its requests. The service is then concerned with any validation, i.e. interacting with the CVS as well as with the PDP. In this proposal, CIS and CVS are colocated and constitute a part of the STS component.

2.6.2.9 Assignment of Arbitrary Authorisation Attributes to Users

Access control solutions in multi-domain computing environments have to take into account the complexity of user attributes that can be potentially used for authorisation. More importantly, if these solutions span different domains of control then types of attributes that are released by distinct Identity Providers are often different (apart from common attribute types such as user identifier). Moreover, some of these attributes, including group memberships, are often meaningful only within the context of a single domain. For example, information about a group membership of a particular user within one domain may not be meaningful in another domain where access control decisions are unlikely to be based on that particular information. A better approach would be to allow assigning attributes that can span users across different domains. For example, users from numerous domains could be assigned a particular group (e.g. within a VO) and access control at different resource providers could be based on such group membership. The actual enforcement and policy checking can still be done locally in each domain.

In highly dynamic multi-domain computing environments, as well as within the context of

the open Web, it is necessary to allow for such flexible attribute assignment. Firstly, it should be possible to introduce new attribute types (such as the previously mentioned group membership) and assign the actual attributes to users across different domains and systems. This would allow these newly created attribute types to be used during policy definition and further in authorisation processes. Secondly, it should be possible to delegate such attribute type creation and attribute assignment to different entities. For example, additional attribute for users from one domain could be controlled by users at another domain.

Systems such as VOMS [135] introduce a new component (VOMS Server) that can be used specifically to meet the aforementioned two requirements in the context of VOs and grid computing. Firstly, VOMS allows to create new attribute types dynamically and assign attributes to users irrespectively of the actual domain that these users reside in. These new attributes can be combined with the actual user identity, as defined by local Identity Providers, and used along those local attributes during access control. Secondly, this new component can be used within a single VO and access to it can be delegated to numerous domains depending on the actual deployment or configuration requirements. VOMS relies on policies to be evaluated and enforced locally in each domain. VOMS is discussed further in Section 3.10.

The UMA proposal, as discussed in this thesis, addresses the aforementioned requirements by allowing the central access control system to interact with distributed IDPs to gather the necessary attributes that can be used during access control processes. Therefore, it is possible to include new types of attributes by establishing a link between the discussed system and new IDPs. Similarly, users from various Web domains can use the discussed proposal to define which attributes are necessary, irrespectively of the domains of IDPs for those attributes. Enforcement of policies is done locally but decision making is delegated to a central component.

2.6.2.10 Security of Access Control Systems

Access control systems need to be protected against any potential attacks in order to be able to protect resources effectively. Security should be applied to individual components of the access control system as well as to interactions between these components [240].

Single components must be protected against illegal accesses to information that they store or processes that they perform. This relates to access control policies stored by PAPs and the access control decision making process done by PDPs respectively. Interactions between components must be protected to ensure that messages can be exchanged securely.

Access control policies need to be protected as well. Mechanisms used for such protection are typically implementation specific within each computing environment. The approach discussed in [219] presents security mechanisms for the authorisation infrastructure that are based on the

same PEP/PDP mechanisms that protect ordinary resources. This results in the authorisation system being easier to administrate as all access control rules are specified using a single policy language with only one policy administration interface. The security infrastructure can be flexibly managed using policies and does not rely on hard-coded security rules. The authorisation for ordinary resources and for access control policies is based on the same mechanism and single policy language. This results in checks on the security of such system being easier and less error prone. Security for the authorisation system can be introduced in form of a specialised service as discussed in [218].

The UMA proposal that is discussed in this thesis does not address security of individual components. The implementation presented in Chapter 7 does not allow policies to be shared between different users and each policy can be kept private. Scenarios where policy sharing between users is desirable are presented in [243]. Protecting policies is one of the future research directions that is discussed in Chapter 8.2.

Security of interactions between components needs to address confidentiality, integrity and authenticity of messages which are typical requirements for security in Web Service-based computing environments [253]. Such messages as those carrying access request queries need to be protected with encryption and signatures. Encryption guarantees that no information about access control policies or issued authorisation queries is revealed [93]. Signatures guarantee authenticity of messages which is mandatory to ensure that only valid policies are evaluated and that only valid access control decisions are enforced [114]. Apart from message level security, the transport protocol used to exchange those messages needs to be protected as well. HTTP can be secured with such mechanisms as TLS/SSL [193; 172]. Security of the presented proposals is discussed in the next chapters of this thesis.

2.7 Chapter Summary

This chapter presented access control in distributed environments. It provided background information regarding concepts that are used throughout this thesis. It started with an overview of multi-domain computing environments, the open Web and Web 2.0 applications. It presented their specific characteristics that are of interest in the context of access control. Furthermore, this chapter discussed different concepts of access control in distributed environments, access control policies as well as delegated authorisation. It then discussed policy and architecture access control challenges.

Chapter 3

Related Technologies

3.1 Introduction

Access control systems for the Web have been researched extensively and aim to address mostly flexibility [143] or usability [141] challenges. These systems, however, do not appear to be well-suited to the increasingly user-driven Web environment with an ever-growing number of resources and services. Proposals such as eXtensible Access Control Markup Language [97; 113], Security Assertion Markup Language [102; 151] and Identity Web Services Framework [78], or Kerberos [255] have been successfully deployed within single administrative domains or more complex multi-domain computing environments. However, these proposals are not well-suited to environments where individual users, and not entire organisations, are concerned with protecting their individual attributes or sets of resources/services. These solutions, however, provide a strong basis and their various concepts were used in the research presented in this thesis.

Moreover, various proposals for controlling access to resources exposed through Web APIs have been proposed and are in the standardisation process. Most important work includes OAuth 1.0 [200], OAuth 2.0 [202], OpenID Connect [58] and Google's Street Identity [29]. These proposals fit well to the user-driven Web architecture but fail to provide all the necessary characteristics to satisfy requirements identified in Chapter 1. In particular, despite allowing individuals to have control over their data, these proposals are unable to simplify various policy management processes that are discussed in Section 2.6.2.3. Other proposed academic or open-source solutions aim to fit into an open and user-driven Web environment by empowering users with more control over access rights to their data. More notable works include those discussed in [207; 286; 289; 195; 284; 196; 152] and [281].

This chapter discusses the aforementioned technologies for access control and it is organised as following. First, this chapter includes an analysis of various standards and proposals for standards to access control solutions in distributed environments. Furthermore, it provides a review of similar academic research on user-centric authorisation. It also compares some of the presented proposals to the UMAC and UMA solutions.

3.2 XACML

The eXtensible Access Control Markup Language (XACML) [97] is one of the approaches to provide an interoperable solution to authorisation in highly distributed and multi-domain computing environments. XACML is an OASIS standard that aims to specify the following:

1. General purpose access control policy language;
2. Access control decision request/response protocol.

The access control policy language provides syntax in XML for defining action (type of request) rules for subjects (users) and targets (resources). The authorisation request/response protocol defines format of messages and the information flow between enforcement and decision components of the authorisation system. Attributes that are exchanged between components of the system can be encoded using the Security Assertion Markup Language (SAML) [102].

SAML allows to exchange standard authorisation data between components. An example would be when a capability service issues SAML assertions to encode capabilities that are later consumed on the resource side by the enforcement point. The SAML profile for XACML defines how to use SAML to protect, transport, and request XACML schema instances and other information in XACML-based authorisation systems [98].

XACML supports interoperability between domains of trust so that separate components of the security architecture can work together across domains. It aims to replace proprietary policy languages or formats that apply to specific applications only. This enables a consolidated policy view across the entire computing environment. Defining a standard XML-based syntax also aims to address management related issues. It eases development of standard management tools and toolkits that would serve common policy needs. Those tools can be deployed in a centralised manner to reduce operational costs [235].

The request and response protocol that XACML has defined is based on the exchange of XACML context. As depicted in Figure 3.1, XACML-conformant PEPs issue requests and receive responses from XACML PDPs in form of such context. Cases where PEPs cannot issue XACML contexts are not discussed. This can be solved by introducing an intermediate

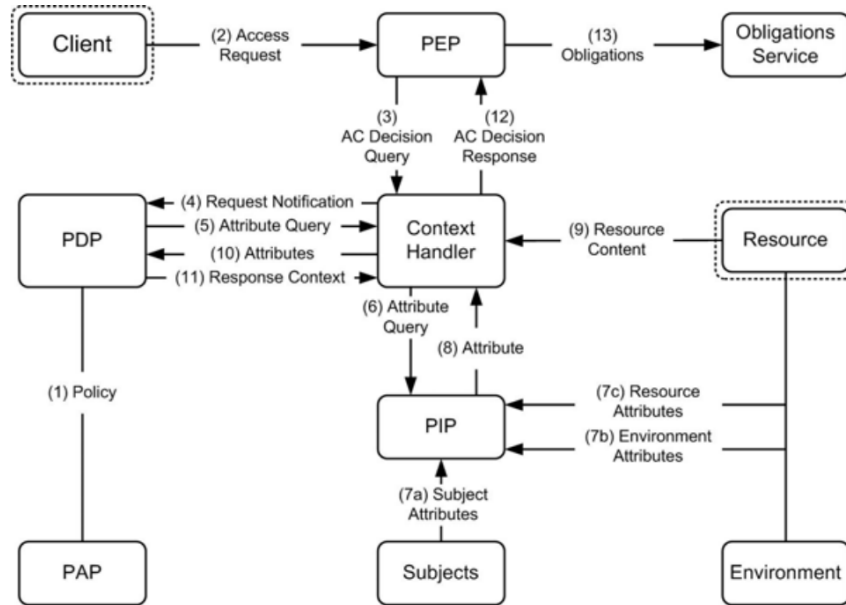


Figure 3.1: XACML data-flow diagram [97].

component, which would convert between the request/response format understood by the PEP and the XACML context format understood by the PDP [97].

The data flow model of XACML, as shown in Figure 3.1, follows the pull model of the authorisation decision query sequence (see Section 2.5). The access request is received by the PEP component which then communicates it to the decision point. PDP evaluates the access request with regards to the applicable policy set, policy or rule and replies with an authorisation decision. To make a decision, PDP components obtain attributes associated with the client, the resource that is being accessed, and the environment in which access request is taking place. Such attributes are retrieved from information points (PIPs).

Decisions, which are made by PDPs, do not only specify whether access should be granted or not but may additionally impose certain obligations on PEPs. Those obligations are an important mechanism of the authorisation system [169]. They allow to define actions which the enforcement point must perform before, with or after an authorisation decision (recall Section 2.6.1.2). Therefore, obligations allow to minimise the list of policies that need to be composed.

With the concept of obligations, administrators can introduce parameterised actions into the policy enforcement stage. An example would be when resources should be encrypted before being returned to the client and the strength of their encryption should depend on some specific attributes of the client, resource or the environment. XACML does not specify how policy obligations should be defined within authorisation decision messages. Therefore, a bilateral agreement between the components of the authorisation system must exist. Enforcement points

need to understand obligations defined within policies stored by administration points [97].

The use of XACML or SAML assumes and requires trust between components of the security infrastructure. However, none of those standards includes provisions to establish or guarantee such trust. SAML, for example, is not concerned with guaranteeing confidentiality, integrity, or non-repudiation of the assertions which are in transit. For the purposes of secure communication, those standards refer to XML Encryption [93] and XML Digital Signature [114] or to other mechanisms provided by the underlying communication protocol and platform [253]. Security is not only provided at the message level with such standards as the above mentioned ones but also at the transport protocol level. When Web Services are used for communication between components of the access control system, then the underlying HTTP protocol is secured with such mechanisms as Secure Sockets Layer (SSL) [193] or Transport Layer Security (TLS) [172].

3.2.1 XACML in Multi-domain Computing Environments

The XACML standard aims to provide authorisation for highly distributed computing environments. This standard along with ongoing research on its extensions fits well into authorisation systems for multi-domain computing environments. This can be considered from different perspectives including the modularity of the system and its ability to span distinct domains of administration, the ability to compose policies or policy sets from distributed sources and providing means of XACML policy administration within the computing environment. Because policies and messages exchanged between components of the access control system are encoded using XML, problems with interoperability between XACML compliant components can be minimised.

The first feature of XACML, which allows its deployment in multi-domain computing environments, is tightly related to its SOA-style architecture where components are exposed and consumed as services. Such services can be integrated in a loosely coupled manner and can be re-used depending on the requirements of the authorisation system that must be provided. Multiple enforcement points can use different decision points of their choice. Those decision points may be located in separate administrative domains and use arbitrarily chosen information and administration points for attribute and policy retrieval. Because policies use a common language in terms of syntax and semantics, they can be easily used in all domains that form a computing environment.

In XACML, policies can be composed of a variety of distributed policies and rules that can be possibly managed by different organisational units [235]. Therefore, rules in such policies may have contradicting meanings. An example would be when there is more than one applicable policy, one that allows access to a resource and one that forbids such access. XACML

describes the use of combining algorithms which support access control decision making process. Decisions can be derived from multiple rules (*rule combining algorithms*) or multiple policies (*policy combining algorithms*). In both cases, it is up to the systems administrator to define which combining algorithms should be used (e.g. *first applicable*, *deny overrides*, etc.). A more detailed explanation of available combining algorithms can be found in [97].

An important feature of XACML that supports its adoption by large scale distributed environments is policy administration which is supported by various XACML profiles. This includes the XACML Administration and Delegation profile [113] which extends policy schema to describe delegation policies. This profile also extends the request context schema to describe administrative requests. A more detailed explanation of this profile can be found in [113].

Another XACML profile to address security needs in multi-domain computing environments is the Cross-Enterprise Security and Privacy [109] profile. It describes mechanisms to authenticate, administrate, and enforce authorisation policies and aims to protect information residing within or across separate administrative domains. Various extensions exist for XACML that address administration of XACML policies as well. The one presented in [219] proposes an XACML-based access control policy model for XACML policies and addresses the problem of rights delegation.

3.3 SAML

The Security Assertion Markup Language (SAML) [151] and Liberty Identity Web Services Federation (ID-WSF) frameworks [78] are federated identity standards that enable identity information-sharing. SAML has been created by the OASIS Security Services Technical Committee [54], while ID-WSF has been standardised by the Liberty Alliance organisation¹ [42].

SAML is an XML-based standard for exchanging authentication and authorisation data between security domains, i.e. between an Identity Provider (a producer of assertions) and a Service Provider (a consumer of assertions). Its aim is to express assertions about a subject in a portable and a standardised way for various applications within and outside of an administrative domain to trust. These assertions can describe authentication statements, attribute statements, as well as authorisation decision statements.

Most typical SAML deployments involve Single Sign-On (SSO) with attribute transfer that takes place at login time, and this is where the ID-WSF defines an architecture that includes a discovery service for locating and gaining authorised token-based access to a person's various identity-enabled services as well as user attributes. Both are able to capture and propagate a

¹On 20th April 2009, the work from Liberty Alliance has been moved to Kantara Initiative, which is where the work on User-Managed Access is taking place.

user's act of consent throughout the authorisation process.

```
1  <saml:Assertion...>
2    <saml:AuthorizationStatement Decision="Permit"
3      Resource="http://example.com/resource">
4      <saml:Subject>...</saml:Subject>
5      <saml:Actions ActionNamespace="http://example.com/abcd">
6        <saml:Action>Read</saml:Action>
7      </saml:Actions>
8    </saml:AuthorizationStatement>
9  </saml:Assertion>
```

Listing 1: Example of a SAML Authorisation Decision Statement.

Additionally, SAML is concerned with providing the solution to access control in environments spanning multiple administrative domains. In particular, it provides a standard way of expressing authorisation decisions (see Listing 1) and a query/response protocol for obtaining such decisions from the IDP (acting as the Attribute Authority). A Relying Party from one domain can query an Attribute Authority from another domain whether access to a particular resource should be granted for a specific subject (either a human being or an application).

To protect the exchange of security information in SAML, both message-level and transport-level security have been provided and both use PKI-based mechanisms. The first mechanism allows to secure communication channel with such protocols as TLS/SSL [193; 172]. The second one allows to sign and encrypt SAML assertions using XML Signature [114] and XML Encryption [93] respectively.

Despite similarities in certain features of SAML and ID-WSF, the UMAC and UMA solutions differ from both proposals. UMA, in particular, is independent from identifier systems and incorporates user-driven policy decision-making in addition to consent alone. UMA uses a similar concept of a query/response protocol for authorisation decisions. It provides transport-level security but can be extended with support for message-level security. In particular, such message level security can be applied to tokens used for authorising requests to resources (refer to MAC tokens used by OAuth 2.0 [55]).

3.4 Kerberos

Kerberos [255; 220] is a well-established protocol developed by the Kerberos Consortium [41], which allows to delegate access control from servers, called Service Servers or Application Servers, to a centralised component, called the Ticket Granting Server (TGS). The flow of the protocol

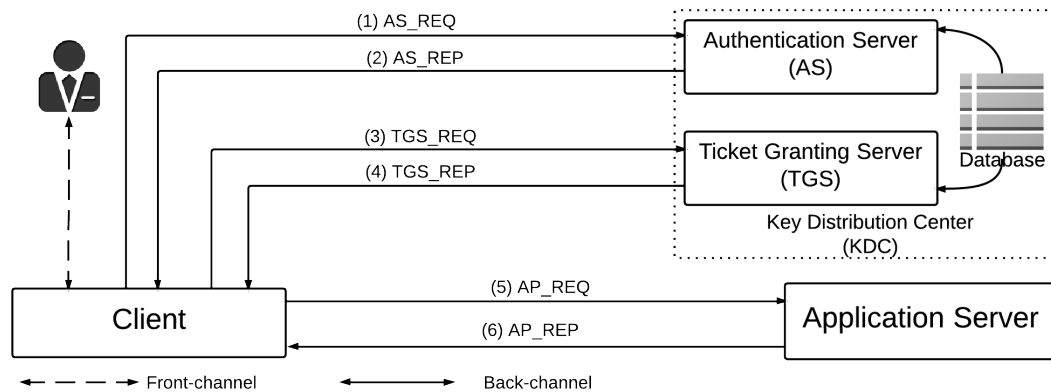


Figure 3.2: Interactions between parties in the Kerberos protocol [273].

is shown in Figure 3.2.

Clients must first authenticate to an Authentication Server (*AS_REQ* and *AS_REP*) and then receive a Service Ticket from the TGS (*TGS_REQ* and *TGS_REP*). Such Service Ticket contains the Ticket Granting Ticket (TGT) that is encrypted using TGS secret key. It also contains the session key (encrypted using the secret key of the requesting user). This ticket can be later used by a client to gain access to a resource on a Service Server which validates this ticket locally based on a session key that was used to encrypt it (*AP_REQ* and *AP_REP*). Only then access to a resource is granted.

Kerberos allows to centralise authorisation for various services but it requires a well-defined point of authority, relationships between services to be pre-established and relies heavily on symmetric cryptography. As such, it is ill-fitted for the open and dynamic Web environment. Moreover, in the current form, Kerberos does little to empower users with better control over access to their online resources. The recent proposal of OAuth 2.0 support for Kerberos [118] seems to allow this protocol to be easily extendable to the Web. However, this approach inherits the weaknesses of OAuth 2.0 (see Section 3.6).

3.5 OAuth 1.0

OAuth 1.0 [200] is one of the early proposals to address authorisation between Web services that attracted much attention in the Web community. It has been based on proprietary standards, most notably Flickr Auth [13], Google AuthSub [17] and Yahoo! BBAuth [91], as discussed in [122]. OAuth 1.0 allows a Resource Owner (RO) to share data between two Web applications, one being a Server and the other being a Client. Access to data is authorised by RO at the Server side which results in an access token being issued to a Client. As a result, Client does not learn credentials of RO and is able to make authorised access requests to selected resources

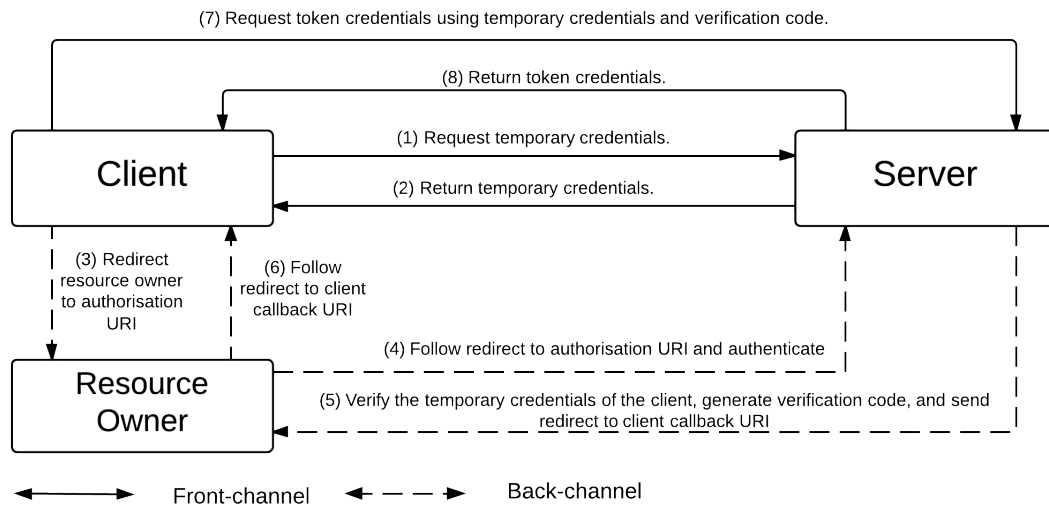


Figure 3.3: OAuth 1.0 protocol flow [200].

or services at a Server using this acquired scoped token. Commonly, each Web application acts as an independent OAuth 1.0 server. The OAuth 1.0 protocol flow is visualised in Figure 3.3.

A crucial part of the OAuth protocol is the concept of *scopes* and *authorisation page* (sometimes called *consent page*). A scope combines a resource (or a group of resources) with an operation (or a group of operations). For example, a scope *calendar_view* can allow for read-type access to a calendar feed, while *documents_manage* can allow for both read-type and write-type access to a set of documents stored at a Web application. An authorisation page is presented to the user and provides them with the necessary information to make a meaningful decision whether access should be allowed or not. This information includes the name of the application as well as the list of scopes for which access will be granted. The concepts of scopes and authorisation page are used in the UMAC and UMA proposals.

As discussed in [200] and [122], OAuth uses three kinds of credentials: *client credentials*, *temporary credentials*, and *token credentials*. Client credentials are obtained during a registration step and are provided to the application out of band. Temporary credentials are requested by the application before obtaining the actual authorisation and eventually token credentials from the Server. These credentials are not RO-specific but are used solely to identify requests and authorisations. Token credentials are issued based on RO authorisation and are used to access resources (Listing 2).

Authenticity and integrity of requests in OAuth 1.0 is achieved with digital signatures (refer to Section 3.4 of the OAuth 1.0 specification [200]). Clients, when issuing requests to Servers, compute a signature of the HTTP request using a keyed-hash message authentication code (HMAC [221]) which calculates a *Message Authentication Code (MAC)* in combination with

```
1 GET /resource HTTP/1.1
2 Host: example.com
3 Authorization: OAuth realm="Example",
4     oauth_consumer_key="9djdj82h48djs9d2",
5     oauth_token="kkk9d7dh3k39sjv7",
6     oauth_signature_method="HMAC-SHA1",
7     oauth_timestamp="137131201",
8     oauth_nonce="7d8f3e4a",
9     oauth_signature="bYT5CMsGcbgUdFH0bYMEfcx6bsw"
```

Listing 2: Example of HTTP request, carrying token credentials, sent by the client application to the server [200].

a secret cryptographic key. In particular, OAuth 1.0 supports HMAC-SHA1 where the client secret and token secret are used as a key and the SHA-1 function [262] is used to compute a signature. Unlike client secret, the token secret is provisioned during the protocol flow and therefore requires additional transport level security (i.e. HTTP over TLS/SSL). Furthermore, OAuth 1.0 allows the use of asymmetric cryptography, namely RSA-SHA1, where private/public key pairs are used both by the client as well as the server. In such case, HTTP over TLS/SSL is not necessary.

With scoped tokens being used for accessing protected resources, OAuth 1.0 addresses the password anti-pattern [267], i.e. a user does not have to reveal their credentials to give one service access to protected resources hosted at a different service on the Web. Each OAuth Server used by the user, however, must independently serve an authorisation management function without the possibility of centralised management, defeating requirements identified in Chapter 1. Moreover, OAuth 1.0 requires a person to be present when authorising an access request and does not allow pre-configured policies to be used when making access control decisions. Additionally, client applications are required to be pre-registered with the servers, which limits the use of the protocol in dynamic environments such as Web 2.0. Proposals discussed in Chapters 4 and 5 alleviate these shortcomings of OAuth 1.0.

3.6 OAuth 2.0

OAuth 2.0 [202] is a less complex attempt to solve the problem of authorisation for Web resources to that proposed by OAuth 1.0. In particular, in its base form it only relies on transport-level security (i.e. the use of HTTPS protocol) without message-level security. This reduces the complexity of client libraries and frameworks. It also reduces the application development effort. OAuth 2.0 has been based on the initially proposed OAuth Web Resource Authorisation Profiles

(WRAP) [119], eventually submitted to OAuth Work Group at Internet Engineering Task Force (IETF) [79].

OAuth 2.0 allows a Web application, called a Resource Server (RS), to delegate authorisation to a trusted authority called an Authorisation Server (AS). A Client, when seeking access to a Protected Resource hosted on RS, must first obtain an access token from AS and present this token along with an access request.

Obtaining authorisation from the Resource Owner (RO) can be accomplished using various ways, which are considered as grants for OAuth 2.0. The specification defines four grant types: *authorisation code*, *implicit*, *resource owner password credentials*, and *client credentials*. The first two types are considered a *3-legged* variant, while the latter two types are considered a *2-legged* variant of the OAuth 2.0 protocol.

The 3-legged variant of the OAuth 2.0 protocol involves direct interactions between the following three parties: the Resource Owner, the Authorisation Server, as well as the Client. Importantly, in this variant an authorisation is transferred from the AS to the Client through the RO. For example, a one-time code is transferred from the AS to the Client through the RO's user-agent application such as the Web browser. This authorisation code is later exchanged for an actual access token that allows the Client to interact with the RS. In this variant, the Resource Owner does not share their credentials with the Client.

In the 2-legged variant of the OAuth 2.0 protocol, the Client interacts directly with the AS in order to obtain an access token for a specific RS (and not through the RO). For example, instead of requiring the code to be transferred through the user-agent application, the Client can exchange credentials of the RO directly at the AS in order to obtain an access token from this AS. Importantly, the Client does not have to store these credentials but uses them only to obtain a long-lived access token that is scoped for a particular access type at RS. Because credentials need to be shared with the client, this variant of the OAuth 2.0 protocol should be used only when there is a high degree of trust between the RO and the Client. Additionally, the Client may also use this variant to exchange its own credentials (and not credentials of the RO) for an access token to access resources at the RS. A detailed explanation of different variants of the OAuth 2.0 protocol is given in [202].

The OAuth 2.0 Authorisation Code grant type has been used as the basis for our research on UMAC and it also constitutes an important building block for the UMA proposal. This type allows to obtain the authorisation using the *Authorisation Endpoint*. This grant is presented as part of the entire OAuth 2.0 flow in Figure 3.4 and an example of an HTTP request containing MAC access token in Listing 3.

Access tokens returned by AS to Clients are often short-lived and their validity is implemen-

```

1  GET /resource HTTP/1.1
2  Host: example.com
3  Authorization: MAC id="h480djs93hd8",
4      ts="1336363200",
5      nonce="dj83hs9s",
6      mac="bhCQXTVyfj5cmA9uKkPFx1ze0XM="

```

Listing 3: Example of authorised HTTP request with a MAC access token sent by the client application to the resource server.

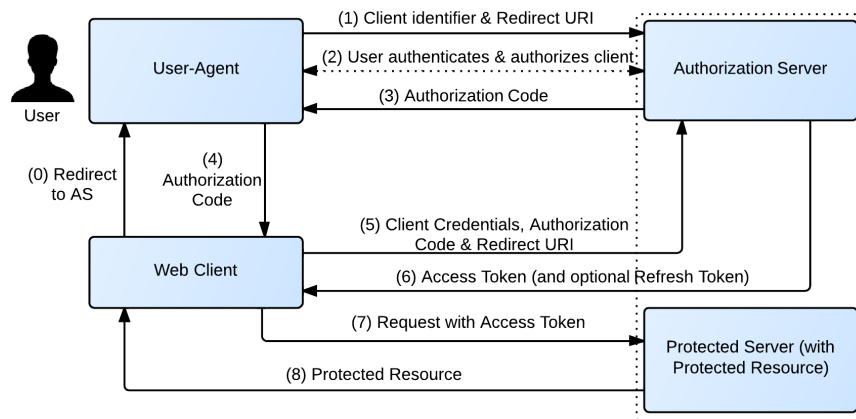


Figure 3.4: OAuth 2.0 - Authorisation Code Grant (former Web Server profile) [202].

tation specific. These tokens are returned from the *Token Endpoint* and can be accompanied by a refresh token. Such refresh token can be used by client applications to obtain new access tokens without the need of going through the authorisation flow. While access tokens cannot be revoked, the refresh tokens can [290].

Similarly to the first version of this protocol, OAuth 2.0 also relies heavily on the authorisation page being presented to the end-user. Such page provides information regarding the application seeking access to resource as well as the list of scopes for which access will be authorised. It is important to note that the protocol allows downscoping, which can be used by the user to limit the list of permissions that the application will be authorised for (despite the fact that the application itself may ask for a broader set of permissions). Unfortunately, such downscoping is rarely implemented (e.g. from existing major OAuth 2.0 providers, only Facebook supports this feature) but may pose security concerns in certain cases [230].

Accessing protected resources in OAuth 2.0 is done by including the access token retrieved from the AS in the request to the RS. A token is usually passed in the **Authorization** header of the HTTP request but can be included in the body of the HTTP **POST** request or as a query in the HTTP **GET** request. OAuth does not mandate the use of any particular token type and

this is dependent on the AS used by the Client. At the time of writing, at least two token types were specified for OAuth, namely Bearer [210] and MAC [55] access tokens. The latter one, in particular, can allow access over insecure channels (e.g. HTTP without the use of TLS/SSL).

Despite allowing the use of an external entity to control access to Web resources, OAuth 2.0 fails to provide certain features required in the open Web environment. In particular, it does not define dynamic introductions by users, requires relationships between parties to be pre-established, and does not define how RS comes to trust AS². Moreover, OAuth 2.0 does not define the communication between RS and AS and in typical scenarios these two components reside in a single administrative Web domain. This requires the hosting application to understand the syntax and semantics of authorisation tokens issued by AS in order to validate them locally, which limits the use of the proposed OAuth 2.0 framework in the open Web. Furthermore, OAuth 2.0 does not allow to demand claims from Clients, which can be perceived as yet another of this protocol's major weaknesses. Therefore, OAuth 2.0 fails to address identified shortcomings (Section 1.1.2) and does not meet all the requirements as discussed in Section 1.1.3.

The UMA protocol, which is discussed in this thesis, relies on the OAuth 2.0 protocol for its interactions between entities of the proposed architecture. It builds on two instances of this protocol. The use of OAuth 2.0 in the UMA model is presented in Chapter 5.

3.7 OpenID Connect

The OpenID Connect 1.0 protocol [58] is the successor of OpenID 1.x [56; 100] and OpenID 2.0 [105] protocols and allows federated authentication on the Web. It is a simple identity layer on top of the previously defined OAuth 2.0 protocol. Most importantly, it allows authentication as well as authorisation. It extends OAuth 2.0 with a new token, called *idtoken*, as well as the *UserInfo* endpoint.

The flow of the protocol differs slightly from the originally proposed flow of OAuth 2.0 in three ways. Firstly, the authorisation request that is sent by the client to the *Authorisation Endpoint* at AS, apart from other requested scopes, has to contain OpenID-related scopes (i.e. `openid` scope, and optionally any of the following: `profile`, `email`, `address`, `phone`). Secondly, the token response from the *Token Endpoint* contains an additional value - the `id_token`, apart from the previously discussed access token and optional refresh token. This `id_token`, which is a cryptographically-signed JSON object encoded in `base64` [39; 38; 37], contains claims about the authentication event and other requested claims. In particular, it defines the authenticated subject's identifier and the authentication method. Additionally, it

²I am a co-author of the recently emerging protocol for dynamic client registration, which is standardised at IETF [274].

clearly states the audience of this token as well as possible authorised presenters (in case the requester acts on behalf of another set of entities, which is common in distributed environments). Another difference between OpenID Connect and OAuth 2.0 is the presence of the *UserInfo Endpoint* which allows the access token to be used for obtaining claims about the authenticated user. The set of claims is standardised in [57] and includes, among others, the name, gender, age, or address of the person.

The OpenID Connect has been used in the UMA proposal to allow users to define their access control policies based on claims and not only identities of individuals. As discussed in further chapters, a policy can allow access to a particular resource/service if the person seeking access can provide claims of a specific type. This is similar to the trust negotiation that is discussed in Section 2.6.1.1, which allows for trust to be built gradually. In UMAC and UMA, for example, access can be restricted based on age or gender of the individual user or it can be based on a proof of "possession" of a particular email address. UMA implementation shown in Chapter 7 supports policies where individuals need to provide a proof of a verified email address.

3.8 Street Identity

The Street Identity protocol is a proposal from Google that builds on top of OAuth 2.0. It is similar to OpenID Connect as it allows for exchange of user attributes between Attribute Providers (AP) and Relying Parties (RP). Attribute exchange is mediated by a Street Identity-enabled Identity Provider³. In this proposal, user attributes can be accessed in a decentralised fashion, i.e. RPs can interact directly with possibly distributed APs to get attributes of the user. The architecture and the protocol of the proposal are discussed in subsequent sections.

3.8.1 Architecture

Street Identity distinguishes the following actors (see Figure 3.5):

1. User;
2. Attribute Provider;
3. Relying Party;
4. Identity Provider.

The *User* is the main actor in Street Identity who dynamically links different parties of the proposal. Firstly, they register their chosen APs with their IDP. Secondly, they authorise

³This section uses the term *Identity Provider* in relation to a Street Identity-enabled Identity Provider.

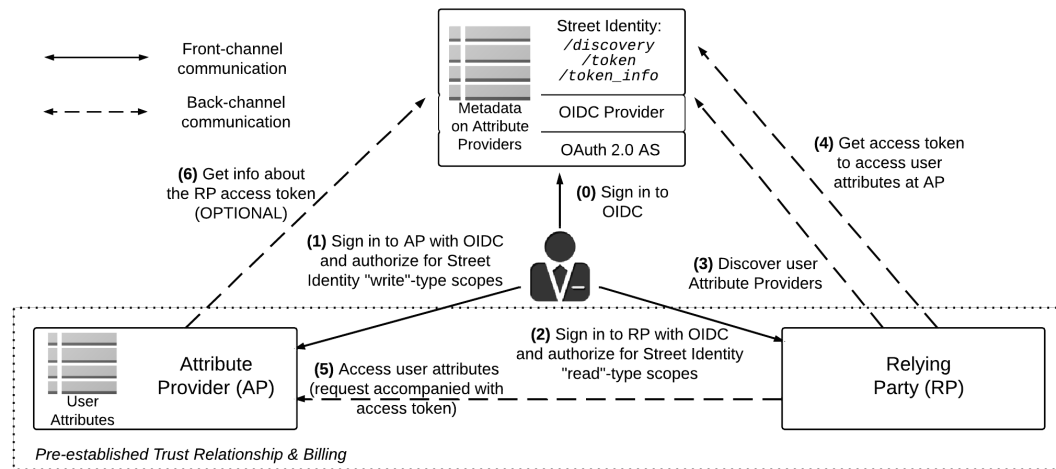


Figure 3.5: The Google's Street Identity proposal.

RPs to access their attributes stored at these APs. The User is assumed to have a pre-existing relationship with the IDP and the AP. Exchange of attributes is usually triggered when the user establishes a relationship with RP.

The *Attribute Provider* is a Web application that acts as a source of trustworthy information about the User. The AP is registered by the User at the IDP only as a source of specific types of attributes. Attributes are exposed through APs programmatic APIs for consumption by RPs. The API exposed by an AP is protected with the 2-legged OAuth 2.0 protocol, or other means, as agreed between the AP and RP. A brief introduction to a 2-legged OAuth 2.0 protocol is given in Section 3.6. A more detailed description is given in [202].

Importantly, a pre-existing relationship between the AP and RP is a requirement. These two types of applications have to be integrated prior to interacting with each other according to the Street Identity protocol. Each access request for an attribute at the AP has to be accompanied by an access token, issued by the IDP, that uniquely identifies the application that acts as the RP. This access token also identifies the owner of the requested attribute. Moreover, such request has to contain credentials (e.g. an OAuth 2.0 access token) that an RP has for this AP. As such, the AP can validate whether access to an attribute should be granted or not. For example, the AP may impose monetary charges on the RP at each request to an attribute. Such billing was one of the requirements for the Street Identity protocol.

The *Relying Party* is an application that acts as a client of information stored at APs. This application is authorised by the User at the IDP to discover APs registered by this user and to learn the location of user attributes. RPs can obtain access tokens for these attributes from the IDP. RPs can be authorised at the IDP to access all or only selected attribute types and such authorisation is given explicitly by the user during the OAuth 2.0 flow. Authorisation cannot

be restricted to selected APs or individual attributes. RPs have to be pre-integrated with APs that they want to interact with (i.e. there has to be a pre-existing relationship between these two applications, as previously discussed).

The *Identity Provider* is an OAuth 2.0 compliant IDP with specific Street Identity extensions (including Street Identity API endpoints) that mediates interactions between RPs and APs. The User can use the IDP to register their APs as well as RPs. RPs can then discover APs registered for a particular user. The IDP is also concerned with issuing access tokens to RPs and these tokens can be used to access user attributes at APs. It also allows APs to remotely validate such access tokens received from RPs.

In order to provide functionality for RPs and APs, the IDP uses standard OAuth 2.0 endpoints as well as newly introduced Street Identity API endpoints. Registration of APs and RPs is done using OAuth 2.0 *Authorisation* and *Token* endpoints. Importantly, such registration only *marks* an applications either as an AP or RP of a particular attribute type at the IDP. This registration is different from the one where an application obtains OAuth 2.0 credentials that are later used to interact with OAuth 2.0 endpoints of the IDP. Other functionality related specifically to Street Identity is provided using the newly introduced endpoints of the Street Identity API. These endpoints are *Discovery*, *Token* and *Token Info* endpoints. The *Discovery* endpoint allows Relying Parties to discover APs that have been authorised by the user as sources of trustworthy information. RPs use the *Token* endpoint to obtain access tokens for specific APs. The *Token Info* endpoint is used by APs to remotely validate these tokens received from RPs. The functionality of the IDP is discussed in more detail when providing an overview of the Street Identity protocol in the next section.

The Street Identity protocol involves a number of access tokens that are used by different actors of this proposal. These tokens are shown in Figure 3.6. The following naming scheme is used for these tokens: $AT_{Issuer-Subject(Audience)}$. The IDP issues the $AT_{IDP-AP(IDP)}$ and $AT_{IDP-RP(IDP)}$ tokens to APs and RPs respectively and these tokens can be used by these applications to interact with the IDP. These tokens are standard OAuth 2.0 tokens (e.g. Bearer-type [159]) which are issued by the OAuth 2.0 *Token* endpoint of the IDP. Furthermore, the RP can obtain $AT_{IDP-RP(AP)}$ tokens which are then used to interact with APs to obtain user attributes. The $AT_{IDP-RP(AP)}$ access token is returned by the Street Identity *Token* endpoint of the IDP and it is a JSON Web Token (JWT) [39], encrypted with JSON Web Encryption (JWE) [37] and signed using JSON Web Signature (JWS) [38]. Furthermore, before an RP can interact with any AP, it has to obtain an access token (or other credential⁴) issued by this AP - $AT_{AP-RP(AP)}$. This token identifies the RP at the AP and allows the AP to include additional

⁴Street Identity leaves unspecified how authentication is done between the RP and AP but the initial proposal would assume a 2-legged OAuth 2.0 protocol between these two parties.

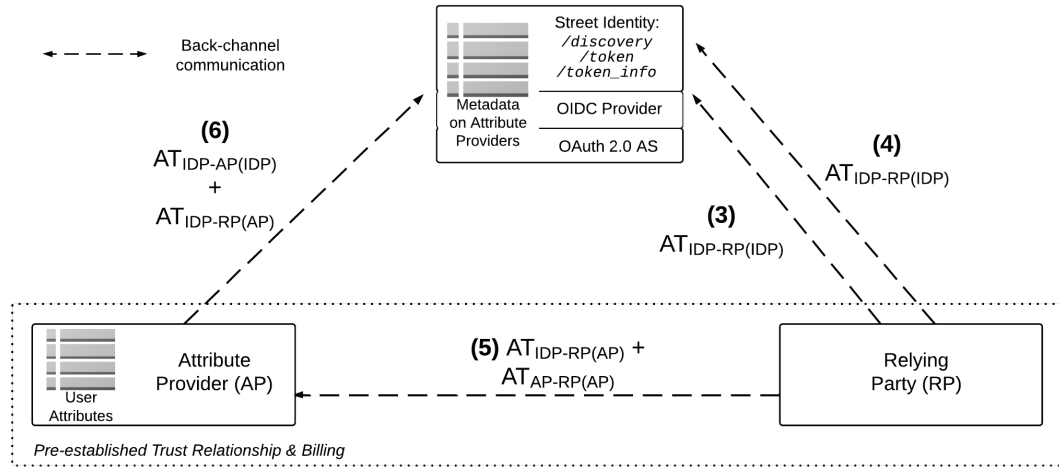


Figure 3.6: Access tokens involved in the Street Identity protocol.

checks in its access control process (e.g. access to user attributes is only given once the AP can charge the RP for such access).

3.8.2 Protocol

The Street Identity protocol is composed of the following steps (as highlighted in Figure 3.5):

1. User registers Attribute Providers at the Identity Provider;
2. User registers Relying Parties at the Identity Provider;
3. Relying Parties discover Attribute Providers;
4. Relying Parties obtain access tokens for specific Attribute Providers;
5. Relying Parties interact with Attribute Providers to obtain user attributes;
6. Attribute Providers validate access tokens before releasing attributes.

In the protocol description that is provided below, various assumptions are made. Applications, acting as either RPs or APs, can interact with the IDP and these applications are already registered as OAuth 2.0 clients (i.e. OAuth 2.0 credentials are issued to these applications). Furthermore, there is an existing relationship between an RP and AP. Most importantly, the RP knows the OAuth 2.0 client identifier which is used by the AP at the IDP. Similarly, the AP knows the OAuth 2.0 client identifier of the RP at the IDP. Moreover, the RP is registered with this AP (e.g. as an OAuth 2.0 client) and understands what information can be obtained as well as what APIs this AP supports. This RP also has the necessary access token (e.g. refer to the

introduced $AT_{AP-RP(AP)}$ for this AP. Moreover, the given discussion assumes that the User is already signed in to the IDP (refer to Step 0 in Figure 3.5).

In the first step of the Street Identity proposal, the User registers AP with their IDP⁵ (see Step 1 in Figure 3.5). Such registration is based on the Authorisation Grant of the OAuth 2.0 flow and requires the user to authorise a client application for one of the existing Street Identity "write" type OAuth 2.0 scopes. Currently, Street Identity supports scopes for the following attribute types: *street address*, *verified age*, and *verified gender*. The corresponding OAuth 2.0 scopes for these attribute types are the following:

1. `https://www.googleapis.com/auth/streetidentity.write`
2. `https://www.googleapis.com/auth/verifiedage.write`
3. `https://www.googleapis.com/auth/verifiedgender.write`

Once an application is authorised by the user for any of the above scopes at the IDP then this IDP issues the $AT_{IDP-AP(IDP)}$ token to this client application which becomes an AP. Authorisation for these scopes is used for a dual purpose. Firstly, these scopes allow to distinguish APs for different attribute types. For example, if an application has been authorised for the `https://www.googleapis.com/auth/streetidentity.write` scope then this application is treated by the IDP as an AP only for the *street address* attribute type. Secondly, if an application has been authorised for any of the aforementioned "write" type scopes, then this application can use the $AT_{IDP-AP(IDP)}$ token to validate tokens ($AT_{IDP-RP(AP)}$) received from RPs by interacting with the *Token Info* endpoint of the IDP.

Importantly, the client application during registration as an AP has to obtain an IDP-issued user identifier and link this identifier with its locally used identifier for the user. This identifier will be used to obtain the correct attributes that are requested by RPs. Importantly, Street Identity protocol, unlike OpenID Connect, does not impose any constraints on the actual API through which attributes are exposed to RPs.

During the second step of the Street Identity protocol, the User registers a Relying Party at the IDP (Step 2 in Figure 3.5). Such registration requires that an application is authorised for one of the Street Identity "read" type OAuth 2.0 scopes. These scopes correspond to the previously discussed "write" type OAuth 2.0 scopes and are the following:

1. `https://www.googleapis.com/auth/streetidentity.read`
2. `https://www.googleapis.com/auth/verifiedage.read`

⁵At the time of writing, Google was the only known Street Identity-enabled Identity Provider.

3. `https://www.googleapis.com/auth/verifiedgender.read`

Once an application is authorised by the user for any of the above scopes then the IDP issues the $AT_{IDP-RP(IDP)}$ token to this client application which becomes an RP to an Attribute Provider (apart from being an RP to the IDP). Authorisation for Street Identity "read" type scopes allows the RP to use the issued $AT_{IDP-RP(IDP)}$ token and interact with the IDP's Street Identity API. Firstly, these scopes define for which attribute types the RP can discover APs. For example, if an application is authorised for the `https://www.googleapis.com/auth/streetidentity.read` scope then this application can only discover APs that are registered for at least the *street address* attribute type. Secondly, if an application has been authorised for any of the aforementioned "read" type scopes, then this application can obtain access tokens for APs of the corresponding attribute types using the *Token* endpoint. Importantly, the RP cannot be restricted to access only individual APs or even individual attributes stored at these APs.

RP has to know which providers are available for a particular user before it can retrieve them from AP. The RP uses the *Discovery* endpoint to learn about available APs for a particular user. The RP interacts with this endpoint using an HTTP `GET` request. An example response, excluding any HTTP headers, returned by the *Discovery* endpoint is provided in Listing 4. This response includes OAuth 2.0 client identifiers for different attribute types.

```
1  {
2      "https://www.googleapis.com/auth/streetidentity.read": [
3          "01234.example.apps.googleusercontent.com",
4          "56789.example.apps.googleusercontent.com"
5      ],
6      "https://www.googleapis.com/auth/verifiedage.read": [
7          "56789.example.apps.googleusercontent.com"
8      ],
9      "https://www.googleapis.com/auth/verifiedgender.read": [
10         "01234.example.apps.googleusercontent.com"
11     ]
12 }
```

Listing 4: Example of response from the Street Identity API Discovery endpoint.

As discussed earlier, the RP can discovery all or only selected APs registered by the user and this depends on the actual authorisation. In the example given in Listing 4, the RP is authorised for all three available attribute types at the IDP that returns information about available APs. The RP can decide which AP to use in case there are multiple APs for the same attribute type (see two APs listed for the `https://www.googleapis.com/auth/streetidentity.read` scope in Listing 4). This decision is made based on pre-existing relationships between the RP and

APs. For example, the RP may decide to use the AP that charges the least money to access an attribute for a particular user. Alternatively, the RP may ask the user to select the AP of their choice. This step of the protocol is visualised as Step 3 in Figure 3.5.

Before the RP can interact with any AP, this application has to obtain an access token for this AP ($AT_{IDP-RP(AP)}$). A token to access attributes is obtained by the RP using the Street Identity *Token* endpoint. The RP interacts with this endpoint by issuing an HTTP `POST` method and providing the OAuth 2.0 client identifier of the AP that this RP wants to interact with. This identifier corresponds to one of the identifiers from the response returned by the aforementioned *Discovery* endpoint of the IDP. Among others, the $AT_{IDP-RP(AP)}$ token provides information about the audience (i.e. for which AP this token was issued) and the user identifier (i.e. for attributes of which particular user this token was issued). Importantly, the user identifier is unique only within the scope of the IDP but it is mapped to a local user identifier at the AP. The step of obtaining an access token from the IDP is visualised as Step 4 in Figure 3.5.

The AP exposes user attributes through its programmatic API and the RP accesses this API with the accompanied access token. Importantly, there is no particular standard on how the attributes should be exposed and there are no requirements in relation to the API supported by the AP. Instead, the RP has to have a relationship with the AP and must understand the API that is used to expose those user attributes. The $AT_{IDP-RP(AP)}$ token is passed by the RP as a query parameter in the HTTP request to the AP, which can validate this token locally (in case encryption has been done by the IDP using a shared secret) or using the Street Identity *Token Info* endpoint. An example response from this endpoint is shown in Listing 5. The AP extracts the user identifier from this token and uses it to get attributes of the correct user (recall the requirement for the AP to map such user identifier, as specified by the IDP, to its local user identifier).

```
1  {
2      "issuer" : "https://www.googleapis.com/streetidentity/v1/token",
3      "audience" : "01234.example.apps.googleusercontent.com",
4      "issued_to" : "11111.example.apps.googleusercontent.com",
5      "user_id" : "0123456789",
6      "scope" : "https://www.googleapis.com/auth/streetidentity.read",
7      "issued_at" : 1368316800,
8      "expires_at" : 1368320400
9  }
```

Listing 5: Example of response from the Street Identity API Token Info endpoint.

RPs and APs need to have a pre-established relationship between each other, which is required for the envisioned billing in the Street Identity proposal (i.e. APs can monetize access to user attributes). Access requests made by RPs to APs have to include additional credentials which are specific for an RP at this particular AP. For example, the RP has to include an access token obtained from this AP (recall the $AT_{AP-RP(AP)}$ token that was introduced in Section 3.8.1) or needs to provide some other credentials issued by this AP. Neither the access token nor the credentials are revealed to the IDP and these are only shared between the RP and AP. Steps of accessing user attributes and validating the $AT_{IDP-RP(AP)}$ token are visualised as Step 5 and Step 6 respectively in Figure 3.5.

3.8.3 Discussion

The Street Identity protocol has been designed and developed at Google after we have developed UMAC and during our work on UMA. Therefore, it has been influenced by these two solutions. In its initial version [70; 53], the protocol would allow for fine-grained access control to individual attributes stored at APs, which is similar to the UMA proposal. Firstly, APs would not only be able to register at the IDP using the standard OAuth 2.0 flow but these APs could also provide metadata about individual user attributes to this IDP. Such functionality required a new endpoint to be exposed by the IDP for APs.

This level of control required an additional screen to be presented to the user during the authorisation phase and this would result in an additional access control policy being created at the AS (apart from a standard OAuth 2.0 related authorisation). Furthermore, the user would be provided with an additional User Interface at the IDP that would fulfil a dual function. Firstly, this UI would allow the user to view registered APs as well as information on individual attributes. Secondly, the user would be able to review access control policies for their attributes stored on distributed APs.

The Street Identity protocol has been eventually simplified, which would limit the control that end users had over attribute release⁶. The user would not be presented with two different authorisation pages, i.e. one page for authorising RP at the IDP and another page to specify which attributes or APs should this RP be authorised for. With the lack of an additional access control policy being created at the IDP, no new User Interface would be necessary. The existing OAuth 2.0 related UI that allows the user to review which applications have been authorised at the IDP for which scopes was sufficient and would allow the user to review their RPs and APs.

The aforementioned simplifications have been done primarily due to two reasons. Firstly, it

⁶Due to the lack of documentation about the changing protocol, refer to the source code revision `a69050731b9c` in [126].

was necessary to simplify the User Experience for end-users who were unfamiliar with the flow involving multiple parties from different administrative domains (i.e. Relying Party, Attribute Provider, Google's Street Identity-enabled Identity Provider). Additionally, it was vital to provide a solution, which could be implemented by developers in their Web and mobile applications without unnecessarily steep learning curve. The complexity of the initial proposal was considered an obstacle for successful deployments.

3.9 Sticky Policies

Sticky policies is another approach of enforcing access control to data that can be spread across numerous applications. As discussed in [269; 163], such policies can be bound to the actual data that is sent to different applications. A user allows an application to have access to specific data (e.g. personal information) based on a policy that is attached to this data. Such data is often encrypted and the decryption key is released only when the application confirms adherence to the terms within the policy. For example, an application may need to interact with a third party component to get access to a key and such interaction is logged for audit purposes (e.g. a user can later review that a particular application has accessed the data and confirmed adherence to the terms contained within a policy).

As discussed in [269], the access to data can be fine-grained and this is based on policy definitions, underlying encryption mechanisms, and a related key-management approach that specifically encrypts data based on the policy. A third party is responsible for mediating access to data, checking for compliance to policies to release decryption keys. This provides users with fine-grained control over access and usage of their data in the open environments, such as the Web.

Authors in [269] give an example of the use of sticky policies where medical data is passed between institutions. In this example, a health-care record is passed from a hospital to a research institution and later to a research team. In such a record, medical results or personal information (e.g. name and address) are encrypted. A sticky policy would be associated with such record describing how parts of this could be used. For example, a policy may state that information can be released only to research teams, it should be deleted after three years, and a notification should be sent to the patient every time the information is passed on from one system to another.

3.10 VOMS

Virtual Organisation Membership Service (VOMS) [135; 136] is a system that allows for flexible assignment of attributes to users. Such attributes can be later used in access control decisions made by resource or service providers. VOMS introduces a new component (VOMS Server) that can be used specifically to meet two important requirements in the context of Virtual Organisations and grid computing. The VOMS Server acts as a central entity that provides its functionality to different domains of control (within a Virtual Organisation).

Firstly, VOMS allows to introduce new attribute types (e.g. group membership) and assign the actual attributes to users across different domains and systems. These newly created attribute types can be used during policy definition and further in authorisation processes within the computing system. New attribute types can be created dynamically and can be assigned to users irrespectively of the actual domain that these users reside in. Importantly, these new attributes can be combined with the actual user identity, as defined by local IDPs, and used along local attributes during access control. Secondly, VOMS allows to delegate such attribute type creation and attribute assignment to different entities. The VOMS Server can be used within a single VO and access to it can be delegated to numerous domains depending on the actual deployment or configuration requirements. For example, additional attribute for users from one domain can be controlled by users at another domain. VOMS relies on policies to be evaluated and enforced locally in each domain.

3.11 Locker Project

A system named Locker [80] (recently commercialised as Singly [73]), describes itself as a container for personal data, which gives the owner of the data the ability to control how this data is protected and shared. Locker acts as a central system that retrieves and consolidates data from multiple sources, including Google, Facebook or LinkedIn applications, among many others [45]. As such, it creates a single view of online data that the user stores at their Web applications. Data is retrieved from applications using connectors implementing a particular protocol provided by an application (e.g. OAuth 2.0 for Google and Facebook; OAuth 1.0 for LinkedIn). This data is later stored in Locker to form a collection (aggregation of common data types like photos, contacts, etc). An actual copy of the data is retrieved to Locker and not just a reference. Collections are further exposed to applications through a RESTful API to which access is protected based on user's requirements.

Locker, like UMAC and UMA proposals, recognises similar shortcomings in existing access control models in today's Web 2.0 environment. In particular, it's objective is to empower

an individual to have the necessary holistic view of their online resources and to share these resources from a central location. Locker envisages the use of such protocols as OAuth or UMA to control access to their resources, which would be accessed through a RESTful API.

However, Locker provides access control for data that it actually stored within its platform and not data distributed on the Web. Therefore, despite giving users the convenience of managing access rights to their Web resources from a central location, it fails to scale with the ever growing number of those resources. Firstly, having two copies of the same resource in different locations requires synchronisation techniques in case such resource is changed. Secondly, despite having a holistic view of data sharing at the Locker platform, the user does not gain any insight into sharing done via applications that store original resources. For example, a particular photo can be shared from Locker with a selected group of users but the same photo can be accessible at Facebook by a different group. Moreover, with the growing number of resources on the Web, having to store a copy of each resource seems to be ineffective.

3.12 Menagerie

The Menagerie system proposed in [195] recognises the problem of data being distributed across multiple Web applications residing in distinct administrative Web domains, which need to be accessed and managed by individual users. In PC-based systems, access to such data would be simplified, irrespective of the application being used. Additionally, a single access control mechanism could be applied to that data. However, on the open Web, this data is isolated in separate Web applications as seen in Figure 3.7, each implementing its own authentication, authorisation and sharing schemes. Authors recognise the need for a new service interface to be defined and adopted by Web applications, similarly to trusted layers that exist in modern operating systems.

The system proposed by the authors addresses the challenge of data organisation and management, data manipulation and processing, as well as protected data sharing on the open Web. In data protection, emphasis is put on allowing individuals to aggregate and share collections of distributed objects with other individuals or groups of individuals. It differs significantly from the previously discussed Locker project, because it puts emphasis on objects remaining stored and managed by their respective Web services and only aggregating pointers to these objects.

As discussed in [195], Menagerie facilitates organisation and sharing of collections of Web service objects using two distinct components, namely *Menagerie Service Interface (MSI)* and *Menagerie File System (MFS)*. MSI is an API that facilitates inter-Web-service communication and access control while MFS allows Web applications to access and manipulate distributed data.

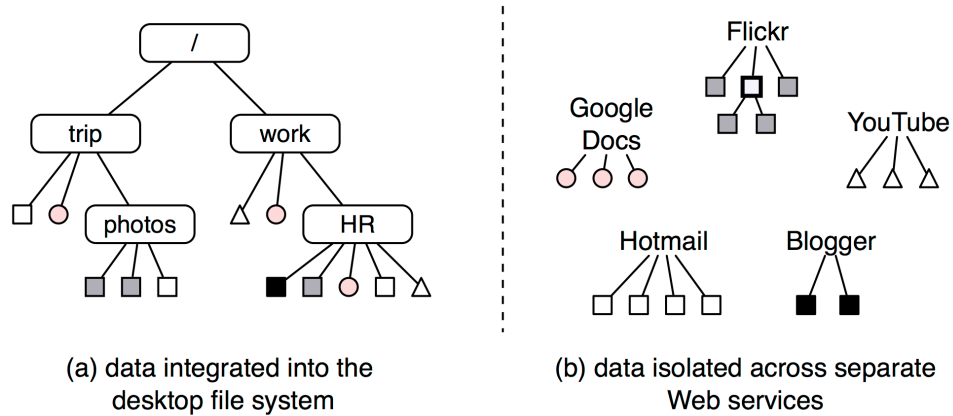


Figure 3.7: Example of data organisation in PC-centric vs. Web-centric world [195].

Such heterogeneous Web services expose access operations through a well-defined API and can be mounted into a local file system namespace. Web applications access this data from the MFS using MSI. This is similar to the proposals discussed in this thesis, which require applications to be enabled for consuming APIs. However, current MSI implementation, as presented in [195], uses an XML-RPC layer [298], while the discussed approaches are based on RESTful Web services. Authors in [195] also provide an overview of implemented Menagerie Group Sharing (MGS) service, which allows for central management of distributed objects, which is similar to the Authorisation Manager that is discussed in this thesis.

Access control is implemented as a "hybrid capability-based protection system". The authors based this approach on the authorised/unauthorised pointer model used initially in the IBM System/38 [146], which merges capabilities with ACL-based authentication. In Menagerie, resource requesters must possess a token to access data and can be subject to additional authentication. For example, *read* type operations can be allowed for clients possessing the capability token, while *write* type operations can require additional authentication from these clients.

In Menagerie, a capability token is an unforgeable token containing the globally unique ID of the (root) object as well as the access rights. It is embedded in the URL which can be shared by users (e.g. using email or by embedding URLs on Web pages). Similar approach is already implemented by existing providers such as Google Calendar [23], which uses secret-links for *read* type operations, augmented with authentication for *write* type operations. Additionally, services have sole control whether access to a resource is granted or not.

As discussed in further chapters, the aforementioned characteristics of Menagerie differ from the approaches that are discussed in this thesis (and from UMA in particular). Firstly, this thesis shows the use of URLs as resource identifiers where access to subtrees of a resource are not allowed (i.e. access has to be explicitly defined for each individual resource or a group of

resources). Secondly, in the presented approaches access control decisions are made by a central component. Implementation of Menagerie presented in [195], includes a *Menagerie Web Object Manager (WOM)*, which lets users organise and share their distributed Web objects. WOM resembles the Authorisation Manager, as discussed in Chapter 4 and Chapter 5.

Capabilities are created using a `create_capa` and can be revoked using `revoke_capa` functions available from MSI. Access rights to resources can be either *open* or *closed*. In the latter case, additional authentication is required from a client, which allows Web services to track users accessing particular objects. This is a requirement for user-centric access control for the Web to allow a user have a holistic view of the applied security controls as well as audit logs.

Menagerie, however, restricts access to Web resources through its Menagerie File System component only. Such restriction does not fit well with the current Web model where data is stored at different applications and exposed through their respective Web APIs. The proposals discussed in further chapters impose certain constraints on developers who must use the proposed protocol, but who are not required to make significant changes to their applications. In Menagerie, however, developers are additionally required to interfere with the business logic of their applications and must use the proposed mechanisms for managing data. This can be considered as an unnecessary level of complexity, which disqualifies this proposal for existing applications.

3.13 Secure Web 2.0 Sharing Beyond Walled Gardens

In [286], authors describe an access control solution that allows users to share content using existing secret-link mechanisms. This solution consists of an architecture and a protocol (refer to Figure 3.8) that aims to solve similar problems to the solutions discussed in this thesis. In particular, authors recognise the problem of "broken" access control in the open Web with such challenges being not addressed as *usability*, *inter-operability*, *granularity of control* and *accountability* [286].

The proposal from the authors is based on two main components. The first one, called `OpenIDemail`, allows users to log in to distributed Web applications using their emails as OpenID identifiers. Translation between emails and standard URL-based OpenID identifiers is made using the Email Address to URL Translation (EAUT). The latter component is called the Open-Policy Provider and provides four different functions. Firstly, it provides a UI for users to define email-based access control policies for Web resources. Secondly, it allows users to associate policies with distributed resources from a single location. Additionally, this component acts as a resource notification service by sending the secret-link of the protected resource to email

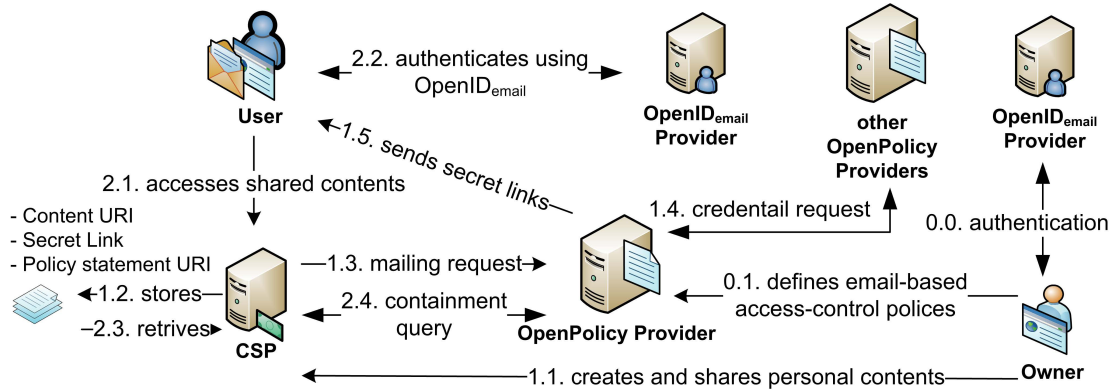


Figure 3.8: The system architecture of the proposed Web 2.0 content sharing solution [286].

addresses following user's policies. Lastly, it evaluates resource clients against policies using proposed distributed inference engine. Importantly, both $\text{OpenID}_{\text{email}}$ and OpenPolicy Provider are user-centric and users can choose their preferred components for these functions. The architecture and the protocol of this proposal are shown in Figure 3.8 and are discussed in more detail in [286].

Resources are shared using secret-links, which the authors identified as most commonly implemented by such providers as Google, Flickr or Facebook. The secret link is generated by the Web application hosting the data and is later combined with the policy by the OpenPolicy Provider (which later sends out this link to clients).

As presented in Figure 3.8 and discussed in [286], the user, acting as the content owner, creates access control policies and associates these policies with distributed content using the OpenPolicy Provider. A client, when accessing data at a Web application, is required to present its identity as well as context information. The latter one includes user-specific attributes, location, date/time of the request as well as user credentials (asserted attributes). This information is used by the Web application, acting as the PEP, to consult the OpenPolicy Provider whether request is valid or not. A decision is made solely by the OpenPolicy Provider, which acts as the PDP and PAP. This component calculates role memberships for the client to match them against the user-defined policies.

The OpenPolicy Provider contains a *Sharing Log* and an *Access Log* as well, which are used for auditing purposes. The first log is used to store information regarding secret-links sent out by the *Distributed Mailing* component. The latter one is used to store information about distributed authorisation decisions (issued by the *Distributed Authorisation* component). Both these logs can be used by the user for accountability and audit purposes and can provide a view of the applied security controls over distributed resources.

The system proposed by the authors is similar to the ones discussed in this thesis from the

perspective of identified challenges that it aims to solve as well as the high-level architecture with a designated component for access control over distributed Web resources. Moreover, it recognises the need for a user-centric access control rather than site-centric model, which is currently deployed on the open Web. According to [286] and similarly to the discussion from the previous chapter, the authors emphasise the need for users to *"have the freedom to choose their favourite providers of their identities, content, social relationships, and access-control policies."*

However, the system differs significantly from the proposals discussed in this thesis in terms of functionality of the components used as well as the protocol itself. In particular, it uses role-based access control as the underlying model for policies. It also relies on secret links. On the other hand, UMA uses identities or arbitrary claims that can be used for access control and also allows resources to be made available through publicly facing Web APIs. This fits well with the current model where applications expose their data through RESTful Web APIs.

Furthermore, this approach targets access to resources made by individuals using their Web browsers. Both UMAC and UMA focus on programmatic access to data by (Web or mobile) applications used by end users. Moreover, the proposed authorisation system can be perceived as tightly coupled with authentication since it requires the `OpenID_email` service in its architecture. Therefore, it may not address various sharing scenarios discussed in [243] and satisfy the requirements, which were presented in Section 1.1.3.

3.14 Secure File System for the Web

A secure file system service for Web applications [207] separates Web data from applications, similarly to existing PC-based systems, where applications access files from the local file system. Authors motivate such separation by pointing out similar challenges that were identified in the previous chapter. In particular, it is mentioned that in the current approach Web applications provide their own data storage with their own-defined Web APIs, which are most commonly incompatible with each other. Controlling access to such data results in access control policies being distributed, which complicates data sharing for individual users.

In the proposed system, files can be stored locally or remotely and can be accessed by Web applications using the system's client component that gets integrated with each application within a user's Web browser. Authors implement a POSIX-like API, which allows for such operations as *open()*, *read()*, *write()*, *truncate()*, etc. [207]. In the proposed approach, the user is free to choose a file system service that best suits their needs. This makes it a user-centric system, which is similar to UMAC and UMA proposals.

Access control is based on capabilities which must be obtained by an application to access

user's data. The user can enforce access control at any time, independent of the application, and this is done using a user interface proposed by the authors. Such interface provides a file browser for the user to delegate access to their files. Selecting a file results in a capability being issued and eventually passed to the application. Closing a file, on the other hand, expires the capability (i.e. revokes it). As discussed in [207], the current implementation only supports single file access, which differs from the proposals that are discussed in this thesis. Additionally, capabilities can be shared between applications with limited ability for their revocation. The discussed implementation has its own access control system for simplicity but envisions the use of such delegated access control as that provided by OAuth [200] or the one discussed in [204].

In the secure file system approach, there is no notion of access control policies. This limits the use of the system in at least two ways with regard to the previously discussed requirements (recall Section 1.1.3). Firstly, authors do not consider sharing to be done without the presence of the data owner. As discussed in Section 3.5 and Section 3.6, this is a limitation that prevents applications from having possibly long-term access to resources that may change over time (e.g. calendar feed, updated resume, etc.). Secondly, the user has no notion of how resources have been shared with distributed applications over time.

In the secure file system model, sharing resources (i.e. files) is done only between a user and their own applications. Authors, however, provide a brief explanation of how sharing could be solved between different applications of the same users or between different users. In the first case, applications could share capabilities while in the latter case the server side component could provide users (and not applications) with these capabilities. Both types of use cases are currently covered by the UMAC and UMA proposals.

The secure file system proposal relies heavily on the trustworthiness of the identities of applications involved in transactions. Additionally, this proposal does not impose any constraints on authentication mechanisms used for applications. Security is achieved mostly through the use of SSL/TLS for communication between the file system (either local or remote) and Web applications. These characteristics are similar to the UMA proposal.

Because the proposed approach requires data to be separated from Web applications, it entails significant changes in existing approaches to building Web applications, which currently rely heavily on having direct access to such data for various reasons (e.g. performance or easy of use). In the current model where resources are distributed on the Web, this proposal does not appear to fully solve the problem of authorisation for Web resources.

3.15 Fine-grained Access Control Policies for Social Networking Applications

In [281], authors discuss data sharing and authorisation for social networks. They motivate their work by the need of fine-grained access control policies for user's data that would meet the user's specific requirements as well as provide them with the appropriate audit information.

In particular, authors state the principles of data sharing and put these principles in the context of such Web 2.0 applications as social networks. These principles are: *"who can see what is clearly the responsibility of the data owner; as one cannot predict a priori what policies particular data owners might wish to impose, support for flexible and fine-grained models is essential; and data owners should be able to know who has seen their data, that is to say that tailored or transformed audits should be available."*

Authors compare the need of sharing data in the open Web with their previous research done in the e-Health field. In particular, they refer to their work in [270], [282] and [283]. Their proposal discussed in [283] describes a mechanism for facilitating access to, and sharing of, healthcare-related data across organisational boundaries. It utilises XACML [97] for policy specification purposes. This is mapped to the context of social networking applications with examples of policies defining access rules for individuals (based on their *name*), *networks*, as well as rules based on such user attributes as *sex*, *gender* or *age*.

The proposals discussed in this thesis take into account the requirements presented in [281] as well as policy types, which are given as examples. In particular, these proposals recognise the need of very fine-grained access control based on arbitrary user attributes. For example, this thesis discusses the use of policies where access is restricted based on the age of the individual user seeking access to a resource. The implementation presented in this thesis also shows integration of social networks into access control policies, where users can share data with their Facebook friends.

3.16 Chapter Summary

This chapter provided an analysis of various standards and proposals for standards to access control solutions in distributed environments. Furthermore, it provided a review of similar academic research on user-centric authorisation. It also compared some of the presented proposals to the UMAC and UMA solutions.

Chapter 4

User-Managed Access Control

4.1 Introduction

This chapter introduces the initial solution to a user-controlled access management in the open Web. This proposal, named User-Managed Access Control (UMAC), fits precisely to the user-driven Web 2.0 environment.

The UMAC approach consists of architecture of services, each with a well-defined role, and access control protocol that specifies interactions between these services. UMAC puts a user in full control of access to their resources on the Web. Unlike existing authorisation systems, it relies on a user's centrally located security requirements for those resources, which may be scattered across multiple distinct Web applications. These security requirements can be expressed in the form of access control policies and are stored and evaluated in a specialised user-chosen component.

In UMAC, a user creates and shares content as usual at their Web applications. However, the user is additionally empowered to flexibly control access to their increasing amount of online resources, using the proposed component. Moreover, a user gets a holistic view of the applied security controls for their online data.

UMAC was initially designed as an external security module for Web applications. This module provided its functionality through a RESTful Web API and acted as a component in Service Oriented Architectures. This proposal was later refined to address the shortcomings and to satisfy requirements that were discussed in Section 1.1.2 and Section 1.1.3 respectively. These shortcomings and requirements were clarified in our further research.

Further work has focused on research and development of the User-Managed Access (UMA) protocol [297], a comparable technology that emerged at the same time as our work on UMAC.

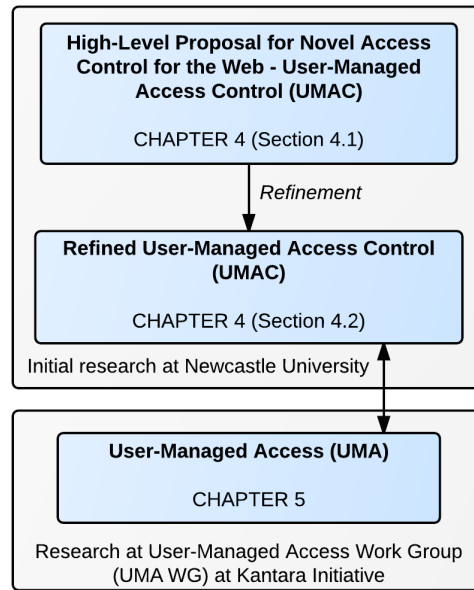


Figure 4.1: Research path on proposals for user-managed access control solutions for the Web.

UMA is presented in Chapter 5. The work on UMA has been conducted at the User-Managed Access Work Group (UMA WG) [86] at Kantara Initiative [40], which includes researchers and professionals from industry and academia. The list of members of the UMA WG is maintained at [87]. UMA proposal has been submitted to IETF [79]. I have been contributing to this workgroup since 2009. In particular, I have been contributing to the use cases analysis, protocol design and implementations. I have been also holding various leadership positions in this work group. Most importantly, since 2010 I have been the vice-chair, use cases editor, and implementation coordinator of this work group.

We did not think it was useful to continue separately the work on UMAC, while concurrently UMA was being developed. Hence, we have focused our research efforts solely on UMA, leveraging our increased understanding of issues related to user experience, design, and implementation of systems similar to UMAC. Figure 4.1 visualises the research path.

The remainder of this chapter is organised as follows. Section 4.2 introduces the initial UMAC proposal to access control by providing its high-level overview. This proposal was published in [242]. The refined UMAC proposal, including its architecture and protocol, is discussed in Section 4.3, which was published in [244]. A prototype implementation is presented in Section 4.3.3. The advantages of UMAC and its evaluation are given in Section 4.3.4. Section 4.3.5 presents possible UMAC extensions and Section 4.3.6 presents its limitations.

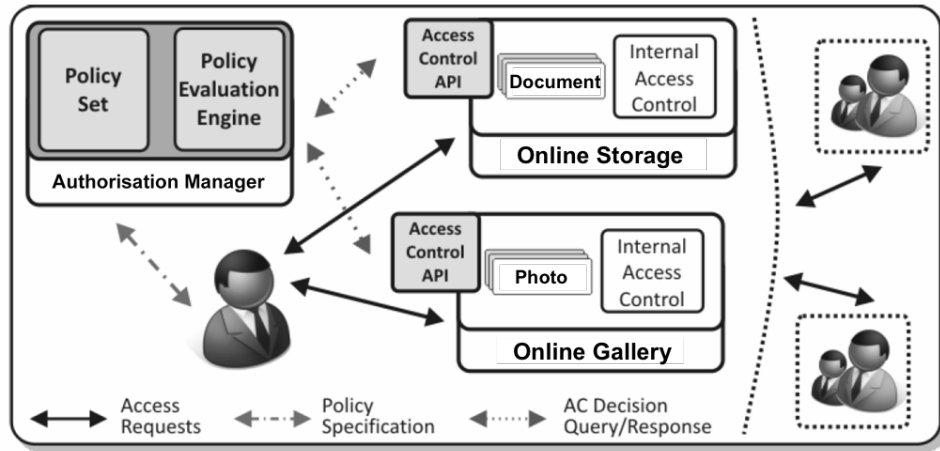


Figure 4.2: High level view of the user-centric access control system for the Web.

4.2 Initial Proposal

User-Managed Access Control¹ is a proposal for a novel user-centric authorisation system for distributed environments, such as the Web 2.0 environment. UMAC is based on the concepts analogous to modern authentication mechanisms, where applications delegate authentication to specialised Identity Providers that are capable of authenticating users for multiple Web applications.

Web applications externalise their authorisation function to a specialised component, called *Authorisation Manager (AM)*. This component was initially named *Security Provider (SP)*, as presented in [241]. Because the aim of the work on UMAC, similarly to the work done by the UMA WG, is to provide users with the mechanisms to centrally control access to distributed data, we have eventually adopted the terminology accepted by this work group in favour of the terminology that was used in our previous architecture [241]. Therefore, the new terminology in relation to UMAC is used throughout this thesis. Security Provider has been renamed to Authorisation Manager.

Authorisation Manager allows users to define security requirements for their distributed Web resources, irrespectively of where these resources are hosted. For example, a user can use the AM to restrict access to their documents stored on an online storage system and to their photos stored on an online gallery service. Access requests to protected resources are subject to access control defined by user policies that are stored and managed at AM. This concept is visualised in Figure 4.2.

¹In the course of the project, we changed the original name of User-Centric Access Control [243; 241] to User-Managed Access Control in order to better reflect the nature of our system with users being in control of access to their Web resources. The change was also dictated by our emerging collaboration with the User-Managed Access Work Group at Kantara Initiative.

4.2.1 Architecture

The generic proposal distinguishes Web applications that have the data, the Authorisation Manager component that secures access to this data, and clients of the data (see Figure 4.2). The user delegates access control functionality from their Web applications to AM using a well-defined RESTful Web API. The AM is a Web application that provides a User Interface for users and an API for applications that want to use its functionality. The AM allows the user to manage their security requirements for their online resources in a uniform way irrespective of the Web application that hosts those resources. Security requirements are encoded in the form of access control policies and the AM makes access control decisions based on those policies, using its provided policy evaluation engine.

A particular Authorisation Manager to be used across applications is chosen and can be optionally controlled by a user. This concept is based on that originally proposed in OpenID [56; 100; 105] where a user chooses their preferred *Identity Provider* according to their requirements and preferences. Moreover, a technically-skilled user may decide to build and use their own AM that implements the proposed protocol and this is also similar to user-centric authentication systems where users may sign in with their custom-built IDPs to their Web applications.

The UMAC approach uses the concept of delegated authorisation (refer to Section 2.4 and [300]) and is related to XACML [97]. In particular, the AM provides functions of the PAP, PDP, and PIP components, which are specified in [97]. On the other hand, Web applications are mostly concerned with enforcing access control policies and act as PEPs.

4.2.2 Interactions

UMAC requires a user to complete two simple steps of the initialisation phase. At first, a user registers an account with their chosen AM. UMAC does not define how an account is created and this step can be completed with various authentication mechanisms. Therefore, the proposal is authentication-agnostic, which makes it usable in various scenarios and deployments with different (and possibly incompatible) authentication protocols used. The user then generates a secret key at the AM using the provided UI and such key is used at a Web application to communicate with this AM. Importantly, the user generates separate keys for each of their Web applications that they wish to integrate with their AM.

In the second part of the initialisation phase, a user configures their chosen set of Web applications to delegate access control for all or part of a user's resources to this component. Such configuration involves providing a URL of a user's AM or selecting one using a graphical User Interface. This approach is based on solutions adopted by existing user-centric identity

federation proposals, such as OpenID. Importantly, the user also configures the Web application by providing a shared secret key that is used for communication with AM.

A user interacts with their Web applications and manages their resources in a usual way. When access control needs to be applied to a resource (e.g. by clicking on a "*Share*" button), a user is redirected to their preferred AM. This is based on the configuration stored by the application and the trust relationship that has been established by the user between their application and the AM (with the use of a shared secret key). At AM, the user can specify either a new access control policy or can apply an already composed policy to a resource. Access control policies allow to specify users who should have access to a particular resource.

Access requests to Web resources are performed as usual by users with their Web browsers. These requests are subject to access control by the AM component. In particular, users have to provide credentials, which were provisioned to them, in order to access protected resources. These credentials are used by the AM to make access control decisions that are later enforced by Web applications. Communication between applications hosting resources and AM is based on the access control pull model and is transparent for clients accessing resources. Such *enforcement model* is depicted in Figure 4.2.

4.2.3 Discussion

UMAC allows the user to choose their preferred Authorisation Manager component to provide the access control function for their set of Web applications. The user then plays the pivotal role in various policy management steps, including those discussed in Section 2.6.2.3 (and presented in [97]). The user has full control of access to their distributed resources and applies centrally located security requirements to protect those resources. Importantly, those requirements allow the user to share their data with other users by providing them with necessary credentials to access online resources.

The authorisation function is externalised from Web applications and encapsulated in form of an easily pluggable Web service accessible by applications over a RESTful Web API. Web applications are only concerned with enforcing access control decisions made by AM. This approach clearly shows that not only functional but also non-functional parts of Web applications can be provided in the form of services that can be used to compose more complex applications on the Web. This approach fits precisely to the SOA paradigm that supports such a decentralised computing model [175; 99] (recall Chapter 2).

Establishing trust relationships between Web applications and AM, and composing security policies is done with the provided UI accessible with a Web browser. Interactions between components of the proposed architecture use HTTP as the transport protocol and conform to the

REST architectural style. In the initial UMAC proposal, transport-level security is provided by requiring the use of TLS/SSL [193; 172] for communication between components. Furthermore, a shared secret key is established between Web applications and the user's AM. The proposed model fits precisely to the user-centred design of the modern Web [251], with different and heterogeneous Web applications that are used for creating and managing data.

The AM may be chosen by the user based on such requirements as available policy languages, policy editors, policy management tools, or the overall User Experience (UX), among other factors. It is believed that there is scope for providers of authorisation services in the open and user-driven Web environment, similarly to providers of identities or attributes. Similar services have been already well-adopted in various multi-domain computing environments, including those forming VOs (refer to Section 2.2.1). On the Web, well-defined interaction sequences and APIs of such services as the proposed access control one, should facilitate their adoption among distributed applications.

Having access control policies stored centrally gives the user means to easily introduce new rules or change existing ones with minimum effort. Additionally, the user can be given a consolidated view of the applied security mechanisms across multiple Web applications, which is discussed as one of the requirements in Section 1.1.3.

Management of access control policies can be simplified through policy reusability as well, i.e. the same policy can be reapplied to different online resources, not necessarily from the same Web application or even the same Web domain. Furthermore, UMAC allows the policy to be applied to a resource even if this resource is moved from one Web application to another. The user does not need to re-establish necessary security rules but can reapply the same policy with minimum effort.

The initial UMAC proposal had various limitations. Therefore, it was considered only as the basis for further research. In this proposal, interactions between different components required manual user intervention. For example, the user would be required to establish a link between the host application and AM by exchanging a shared secret between these applications. Policies that were stored and managed at the AM required that users manually share credentials with other users who would request access to protected resources. Resources themselves would not be registered at the AM and there was no mechanism for applications to register such resources dynamically. Moreover, the proposal targeted use cases where resources would be accessed with Web browsers and this proposal was not tailored for sharing resources through Web APIs. It was also based on a simple pull model and was not easily scalable. Therefore, this proposal was refined and a new version was developed (see next section).

4.3 Refined Proposal

The proposal presented in Section 4.2 constituted the basis for further research. This proposal was refined and its architecture and the authorisation delegation protocol were clarified in order to make it usable in today's Web and applicable to existing Web applications. The refined UMA aimed to address identified shortcomings **S1** - **S4** and to meet formulated requirements **R1** - **R4**, as presented in Section 1.1.2 and Section 1.1.3 respectively. This refined UMAC was published in [244].

In comparison to the initial proposal, two major refinements were introduced:

1. Protected resources can be accessed by Web APIs;
2. Communication between a Web application and Authorisation Manager is based on entitlement model.

The initial UMAC proposal allowed access to protected resources using existing Web browsers controlled by individual users. For example, a user could access a protected document and they were required to provide their issued credentials to access this document. This approach was refined to allow protecting access to resources exposed via Web APIs. Importantly, UMAC focused to support client applications, such as mobile or Web applications (and not User-Agents), to access resources using such APIs.

Secondly, in the initial proposal, communication between applications hosting resources and AM was based on the access control decision pull model and was transparent for clients accessing resources. Such *enforcement model* was depicted in Figure 4.2. The refined UMAC supported the *entitlement model* and allowed requester applications to obtain authorisations to access protected resources similarly to capability-issuing architectures (see Section 2.5.1). The refined UMAC introduced a step of obtaining an authorisation token, similarly to the OAuth WRAP proposal [119]. When the authorisation token is presented to the hosting application, this application validates this token at AM. The AM does not issue an authorisation decision but provides the host with the information regarding permissions associated with this authorisation token. The rationale for this design decision is given in Section 4.3.2.

A similar approach is also adopted by the UMA proposal [297] that based its protocol on WRAP at some point as well (UMA was initially based on OAuth 1.0a but eventually adopted OAuth 2.0). Originally in UMA, the application accessing data would not obtain the token from AM but would rather establish an authorisation state for a scope on an application hosting the data. This state would then be checked by the host application when querying AM for the access control decision. The UMA proposal is discussed in Chapter 5.

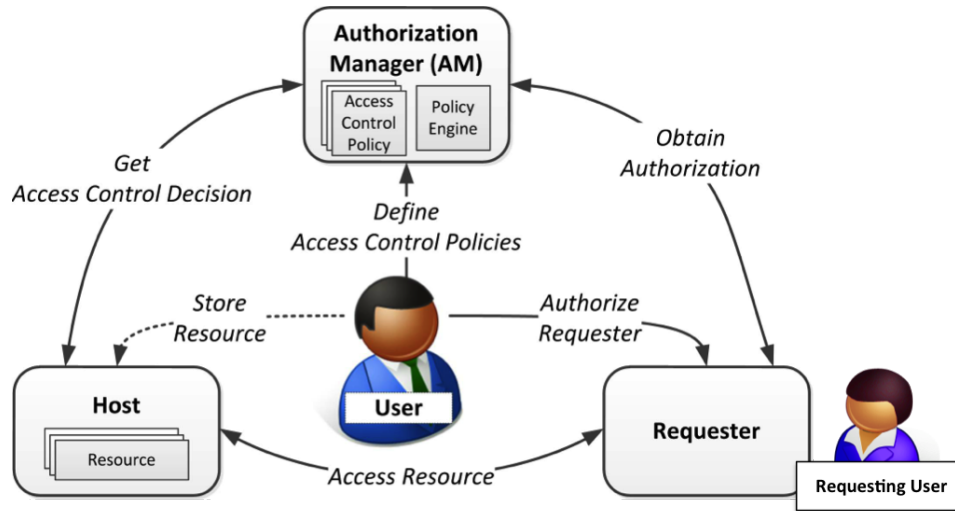


Figure 4.3: Basic Architecture for User-Managed Access Control for the Web [244].

The refined UMAC solution consists of an architecture of services, each with a well-defined role, and an access control protocol that specifies interactions between these services. The architecture is presented in Section 4.3.1 and the protocol in Section 4.3.2.

4.3.1 Architecture

As depicted in Figure 4.3, UMAC distinguishes five main actors that play various roles in this access control system: *User*, *Authorisation Manager*, *Host*, *Requester*, and *Requesting User*.

A *User* is an entity that stores and manages resources using a Web application, called the *Host*. A *User* delegates access control functionality from a single or multiple *Hosts* to an *Authorisation Manager*. The *User* may use the *AM* to define access control policies to protect their resources and may authorise another user, called the *Requesting User* that uses a client application called the *Requester* to access these resources. The *Requester* can issue access requests to protected resources and these requests need to be authorised by the *AM*, which issues authorisation tokens to *Requesters*. Therefore, the entire access control logic is performed by the *AM*, which decides whether access to a resource should be granted or not. A more detailed explanation of each actor is given below:

1. **User:** A *User* is an entity that stores and manages resources using hosts and shares these resources with requesting users. The user also composes access control policies using an *Authorisation Manager* and links these policies with resources. In order to delegate access control from hosts, the user establishes a trust relationship between these hosts and the user's preferred *AM*. In such a relationship, the communication between the host and the

AM is secure, i.e. the integrity, confidentiality, and authenticity of messages between these components is guaranteed.

2. **Authorisation Manager:** The Authorisation Manager is the main component in the UMAC architecture. Its role is to allow the user to define access control policies for their online resources in a uniform way irrespective of the Web application that hosts these resources. It also evaluates these policies before issuing authorisation tokens to requester applications.

The AM provides functionality of PAP, PDP, and PIP components, such as those defined in [302; 97]. Therefore, the general UMAC architecture is related to the one proposed by XACML in multi-domain computing environments [97]. The AM also acts as a Security Token Service (STS) that, following evaluation of access requests, issues authorisation tokens to requesters. This is similar to Kerberos [255].

In the UMAC proposal, the user chooses and controls a particular AM. This concept is based on OpenID where a user chooses their preferred IDP according to their requirements, preferences, or the trustworthiness of this IDP, among other factors. The user may choose their preferred AM based on available policy languages, policy editors, management tools, or the AM's overall UX. More security conscious users may even decide to build their own AMs to protect resources on hosts. A more detailed discussion on user motivations towards choosing the AM is given in [243].

Access control functionality of the AM is used by the user's chosen set of hosts, which delegate access control to AM. These hosts are then concerned with using the AM to obtain information whether authorisation tokens supplied by requesters are sufficient to access a particular resource. The AM provides access control decisions with associated entitlements (permissions) and it is up to the host to enforce them. In the refined UMAC, access control is based on *entitlement model* and such model is discussed in more detail on the UMA example in Chapter 5.

3. **Host:** A Host can be any Web application which allows its users to create or upload and then share their data with other users or services on the Web. Access control functionality of such an application is delegated to AM. Therefore, the host is concerned with enforcement of access control decisions that are issued by AM. As such, the host acts as a policy enforcement point (PEP) [302].

Before the host can offload its access control to the user's chosen AM, the user must establish a trust relationship between these two components. This trust relationship guarantees secure communication between the host and AM using the transport-level security.

Depending on the host's configuration, access control can be delegated to AM either for the entire application, for individual users only, or for individual resources. In a typical setting, it is likely that each user will choose their preferred AM for all of their resources stored on a particular host. UMAC, however, does not restrict such configurations which are implementation specific [243].

4. **Requester:** A Requester is an application that is capable of issuing access requests to resources that are stored on hosts and that are protected by AM. The requester obtains the necessary authorisation token from AM and presents this token to the host when issuing access requests to protected resources. Such token may need to be obtained only once and can be used for multiple subsequent identical access requests to the same protected resource (or the same set of resources) indefinitely (or for a specific period of time) or the token may need to be obtained for each single access request. This is implementation specific and depends on the actual deployment environment. Firstly, the token's lifetime can be configured or the token itself can be single use.

A requester can be any Web application, controlled by a requesting user, that accesses resources stored on a different Web application through their Web APIs. An example would be an online photo editing software accessing a photo hosted on an online gallery. An example set of such applications is discussed in this chapter. A requester can be also a Web browser controlled by a requesting user who wishes to access a particular resource stored on a host. In such case, the Web browser has to be able to interact with other components of the proposed architecture according to the defined protocol, e.g. through the use of a plugin.

5. **Requesting User:** A Requesting User is an individual user that uses a Requester (either a Web application or a Web browser with a plugin) to get access to resources shared with this user. Before access to these resources can be granted, the Requesting User must meet conditions defined by access control policies stored at AM. The Requesting User may be the same entity as the User in case resources are shared between applications or it can be a different user when resources are shared between users.

UMAC, similarly to UMA, differs in various ways from classic access management systems, such as XACML, that are based on the concept of centralised access control components. Most importantly, XACML assumes that the resource owner is the host while UMAC distinguishes between these two entities. UMAC is also designed for the open Web where there can be multiple different resource owners at any given host. Furthermore, XACML is tailored towards closed environments where trust relationships between components (e.g. hosts and central authorisation

systems) can be pre-established while UMAC allows such trust relationships to be established dynamically by the user. Importantly, nothing in UMAC (and UMA) prevents trust relationships between different entities to be pre-established out of band but this thesis does not focus on such setup. UMAC, similarly to UMA, also does not mandate the use of any particular policy type and allows the user to protocol interactions [180].

Figure 4.4 presents a conceptual difference between a user-managed access control proposal and existing solutions. Such comparison was originally prepared for UMA and published in [245]. However, this comparison also applies precisely to the UMAC proposal. Importantly, UMAC focuses on individual users who use Web applications to *store* their online resources. These users can *disclose* location of those resources to potential clients. UMAC allows these users and not security administrators to define access control policies for those resources. In particular, requesters must satisfy policies stored at AM, i.e. must meet the *contract* terms, before access can be granted. Hosts, on the other hand, must check *authorisation* from requester applications before access to a resource can be granted.

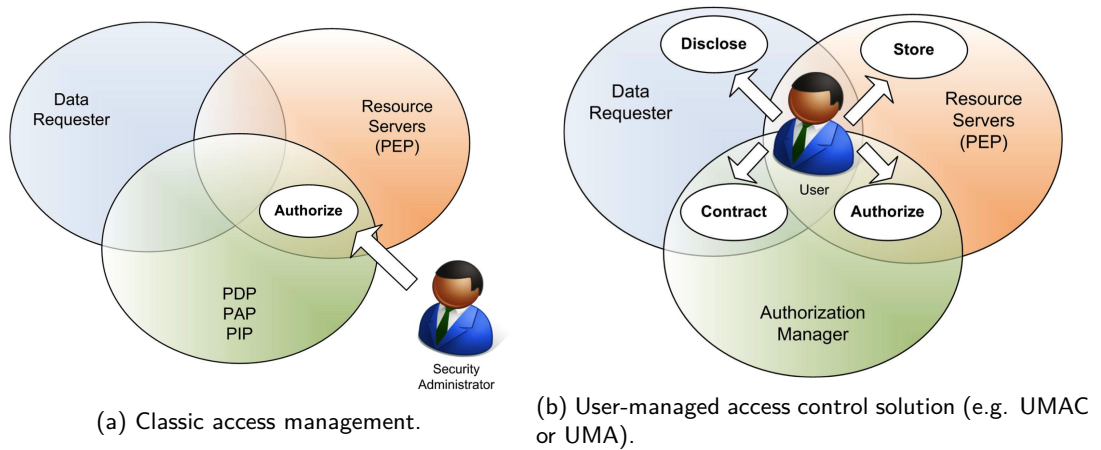


Figure 4.4: Comparison of relations between parties of a user-managed access control solution with XACML [245].

4.3.2 Protocol

A high-level view of the protocol defining interactions between all parties of the UMAC system is depicted in Figure 4.5. The protocol consists of the following steps: (1) *Delegating Access Control*, (2) *Composing Policies*, (3) *Acquiring Authorisation Token*, (4) *Accessing Protected Resource*. The protocol flow for *Subsequent Access Requests* (5) is also discussed. This section presents the steps but without all the details, because the protocol will be superseded by UMA. UMA is presented in detail in Chapter 5.

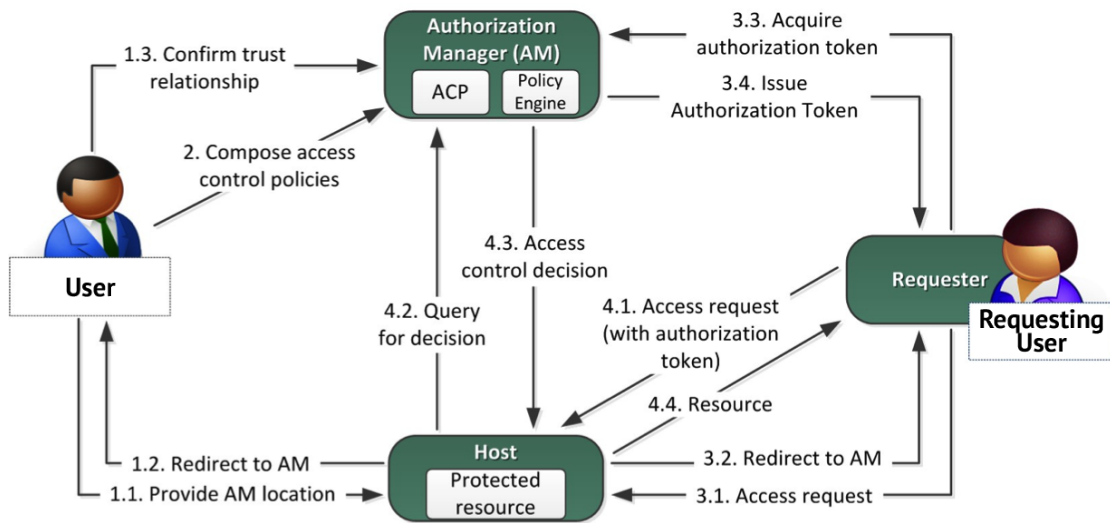


Figure 4.5: High-level overview of the UMAC protocol [244].

1. **Delegating Access Control:** During this step, the user establishes a trust relationship between the host and the Authorisation Manager (Figure 4.6). This is done by collecting the user's consent at AM and providing a host with a *delegation token*. The host can use this token to protect resources using this AM. At this point the host becomes the PEP while the AM acts as the PDP/PAP/PIP component.

The delegation token is a bearer token opaque to the host. It allows this host to use the functionality provided by AM. It has similar characteristics to single-sign-on (SSO) cookies used in Web browsers. UMAC proposes that the size of such token is at least 128 bits.

Importantly, the user trusts the host application that it will delegate access control to the AM. Furthermore, the user trusts the AM that it will issue an access token to the host and that it will provide access control for resources stored at a host. Trust between the host and AM is created dynamically by the user.

In UMAC, the user has to manually register resources at the Authorisation Manager. This proposal does not support dynamic resource registration. Such registration has been introduced in the UMA proposal.

2. **Composing Policies:** The user stores and manages resources on the host as usual. The way access control is exposed to the user depends on the application but is typically achieved by providing a configuration menu. Such menu is often accessible by clicking on a security-related link associated with a resource (e.g. *"Share"*, *"Sharing Settings"*, *"Protect"*, etc.).

Because the host now delegates its access control functionality to AM, then after clicking

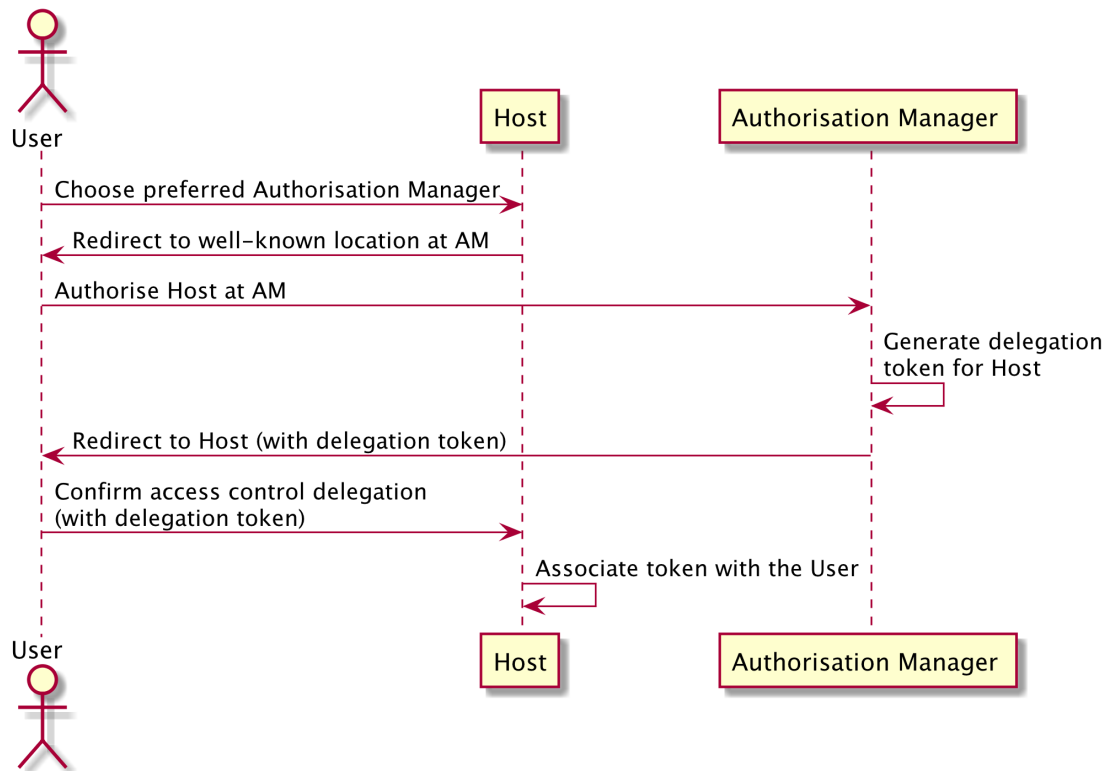


Figure 4.6: UMAC Step 1: A User establishes a trust relationship between a Host and AM.

on such a link the user, instead of defining access control policy at a host, is redirected to this AM. The user can then associate a resource or a group of resources with an already existing policy or may define a new policy as necessary.

Access control policies in the proposed system can be arbitrarily complex and may require requesting users to authenticate themselves or submit necessary claims. In the first case, policies require the requesting user to authenticate to the AM before access can be granted. In the latter case, there can be a more complex process of negotiating access to a resource based on claims provided by requesting users. This process involves requesting claims from the requester and evaluating the submitted claims at the AM. The concept of authorisation is discussed in Section 2.6.1.1.

There can be multiple different policies associated with a resource or a group of resources. With multiple policies in place, it is common for policy conflict resolution protocols to be used. UMAC currently supports positive statements in access control policies. Therefore, this system does not require such protocols to be adopted.

3. **Acquiring Authorisation Token:** In order to access the resource protected by AM, the requester has to issue an access request accompanied by an authorisation token. This is

the push model that was discussed in Section 2.4.1. The authorisation token is bound to a particular resource or a group of resources (scope), a method that needs to be executed on these resources, and the requesting user. This token must be obtained from AM. The requester learns about the AM that protects a resource by issuing an access request to the host, which replies with AM location in the **WWW-Authenticate** header.

Requesters obtain authorisation tokens from AM by providing information regarding the original access request to a host. The AM uses this information contained in the request to find applicable policies and it evaluates the access request against these policies.

Policy evaluation is independent of the protocol itself. UMAC allows AM to support policies based on identifiers or claims. Therefore, the requester may need to engage the requesting user in the authorisation phase (either providing its credentials or submitting self-asserted claims, as depicted in Figure 4.7).

The AM issues an authorisation token to the requester based on policy evaluation. The authorisation token is opaque to the host and to the requester. Therefore, the host must use the AM for the validation process.

Using the push model in UMAC allows the host to be only concerned with receiving a token that must be checked for permissions at the AM. Firstly, the AM only has to interact with the requester to communicate what information is necessary to satisfy a particular policy and the host does not have to participate in this process. Moreover, the requester itself interacts with the AM and supplies the necessary information in order to obtain authorisation for a particular resource or a set of resources. This process of obtaining authorisation may involve multiple exchanges of messages with this AM. Therefore, the pull model would be inefficient in such cases and would require the host to implement additional functionality, which is now placed at the AM.

4. **Accessing Protected Resource:** A requester uses the authorisation token in access requests to resources at the host. Tokens are bound to a specific resource or a group of resources as well as an access method. Therefore, these tokens can be only used to access specific resources.

The host extracts the token from the access request and starts the token validation process - either remotely using AM or locally, if the decision for such request is cached. The AM evaluates access requests and issues decisions: either *"permit"* or *"deny"*, with the attached set of permissions for a token (i.e. entitlements are communicated by AM to the host). Based on the decision as well as attached permissions, access to a resource is granted or not. The host may choose to cache access control decisions issued by AM.

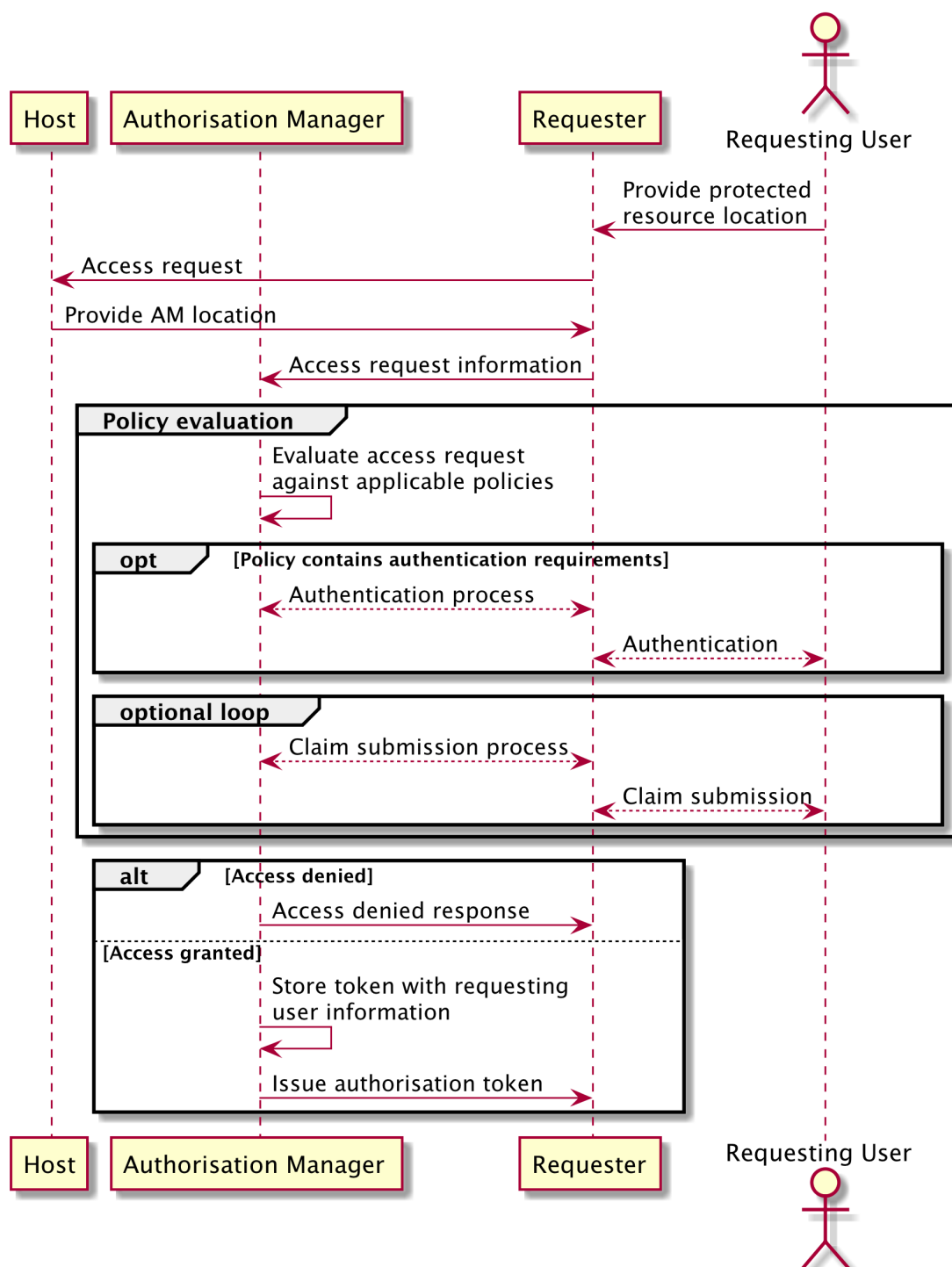


Figure 4.7: UMAC Step 3: A Requester obtains authorisation token from AM.

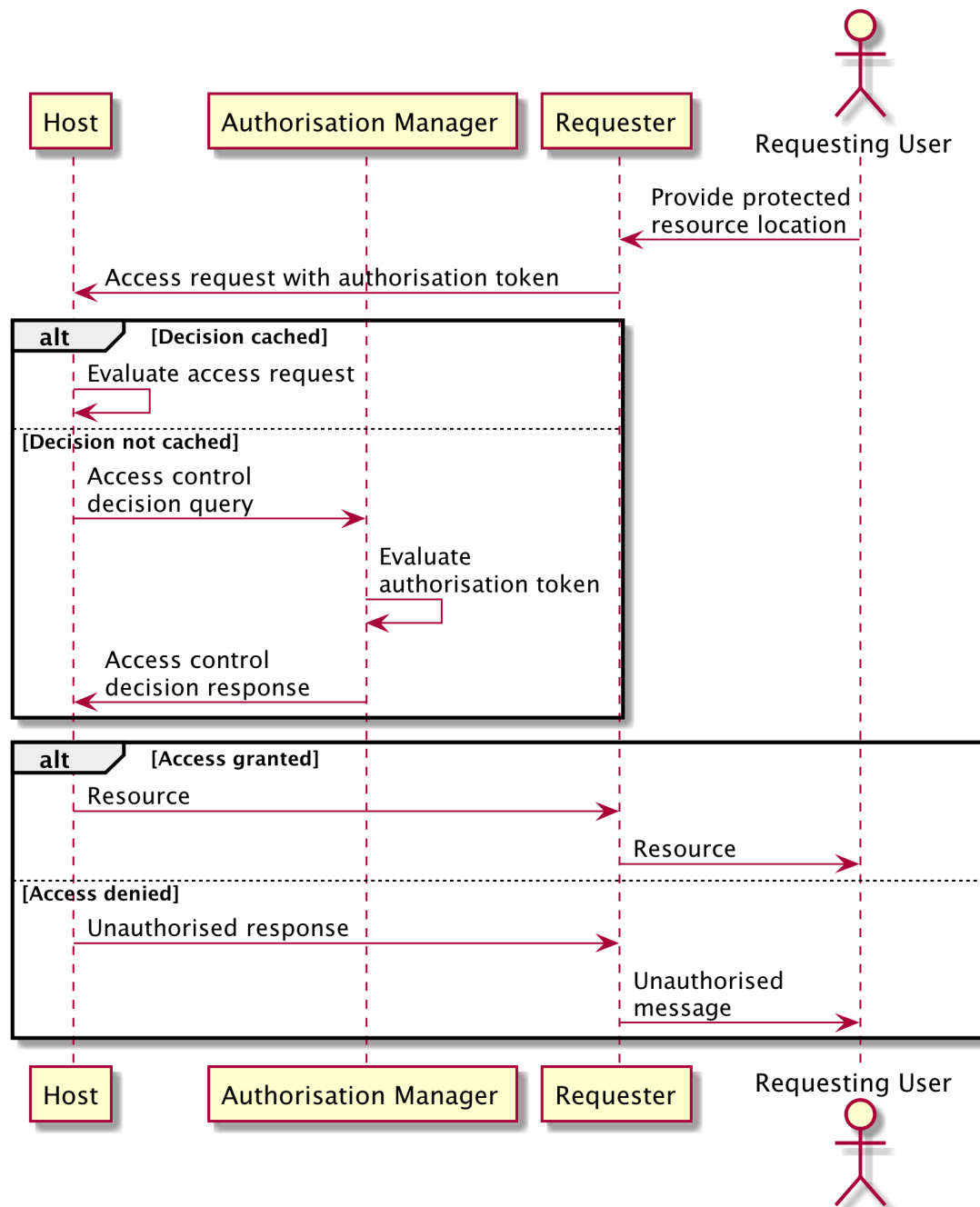


Figure 4.8: UMAC Step 4: A Requester attempts access with authorisation token.

5. **Subsequent Access Requests:** Subsequent requests to protected resources do not have to follow all the steps of the protocol and can be simplified if the requester is already in possession of the authorisation token. The previously obtained decision can be reused by the host, which does not have to validate tokens at AM.

The flow presented in Figure 4.5 as well as figures for individual steps of the protocol deliberately does not include details on the authentication process between the parties of the UMAC system. UMAC assumes that this process can be completed with existing technologies. For example, the user could authenticate to the host or AM using username and password or OpenID and the requesting user could authenticate with OpenID Connect credentials or SAML assertions [151].

4.3.3 Implementation

A simple prototype of the Authorisation Manager has been implemented. This AM allows the user to compose access control policies and apply them to a set of resources hosted on different Web applications. The AM provides both Web-based and RESTful interfaces and supports creating, updating, deleting and reading policies which are persisted in a datastore [19]. The Web-based interface can be used by the user to manage policies and the RESTful interface is used for interactions with hosts and requesters.

The prototype AM implementation allows the user to create a set of general policies from which a single policy can be applied to a resource or a group of resources. In addition, a resource or a group of resources can be linked to a specific access control policy. For example, the user may compose a general policy which defines identities of requesting users that should have access to these resources. The user could also compose more specific policies and further restrict access to particular subsets of protected resources.

Evaluating access requests against access control policies is performed by a custom-built policy engine and the resulting decision can be either *"permit"* or *"deny"*, with the attached permissions for the token. First, the engine evaluates the request against the general policy defined by the user for the group of resources to which a particular resource belongs. For example, the AM can evaluate a policy for the directory of files. If the evaluation results in a *"deny"* decision then no other policy is processed. Otherwise, the engine checks whether a specific policy is associated with the resource, evaluates the access request against this policy, and produces a final decision. For example, the AM can evaluate a specific policy that is associated with a single file.

A custom policy engine was used instead of an existing one because at that time the re-

search focused on the architecture and the protocol of a new access control solution and not on authorisation policies. Development of a custom engine was not considered a challenge and such engine would simplify testing of the UMAC solution. Furthermore, having a custom engine was efficient during its integration with UIs for setting access control policies at the implemented AM.

Furthermore, two prototype hosts have been implemented - the Secure File System and the Online Photo Gallery. The first one is an online file system accessible over the Web where the user can upload arbitrary files and create an arbitrary directory structure. The second application allows the user to upload photos and create photo albums. In addition, it allows the user to edit their photos (resize, rotate, crop, etc.). Thus, this application also acts as a Web-based photo editing tool.

Both developed hosts have a built-in access control functionality allowing the user to define access rights to their resources. The user, however, can configure both applications to delegate access control to the implemented prototype AM. The user can then compose policies at the AM and can link these policies to resources hosted on implemented Web applications.

Both implemented Web applications can act as requesters. The photo gallery can access photos hosted on the online file system depending on access control policies that are defined for these photos. As such, the user can store photos on the online file system and can upload them directly to the photo gallery to produce albums or to edit photos as necessary. Analogically, the file system can access photos hosted on the photo gallery, e.g acting as a backup service for these photos.

Additionally, a requester application, in form of a simple Web client, has been implemented. The application is able to interact with two host applications through their RESTful APIs. It uses the discussed protocol to obtain authorisations to access protected resources.

The prototype software was implemented in Java and can be deployed to the Google App Engine (GAE) platform [21]. The discussed applications make use of the rich set of APIs provided by the GAE platform to achieve the described functionality. Data in these applications is stored in the datastore provided by Google App Engine [19].

All three client applications, i.e. Online Gallery Service, Secure File System, and a requester application, have been used as a basis for further development of the software that would implement the UMA proposal. In particular, newly developed UMA frameworks were later added to each application (substituting any UMAC-related code). Moreover, the software itself was extended and its user interfaces were improved. Therefore, the implemented client applications are discussed in more details in Chapters 5, 6, and 7 on the example of the User-Managed Access proposal.

4.3.4 Advantages

The UMAC proposal for the open Web environment and its implementation have several advantages over existing solutions. In particular, UMAC addresses the identified shortcomings discussed in Section 1.1.2 by meeting the requirements formulated in Section 1.1.3.

1. Access control functionality is externalised from Web applications and is provided in the form of a service, called Authorisation Manager. Its configuration capabilities depend on the particular AM used by the user and do not depend on the actual Web application. Therefore, the user can decide to use AM that satisfies their particular needs and provides the required level of sophistication. This characteristic of UMAC meets requirement **R1** as the AM can provide a rich functionality which could support complex Web transactions irrespectively of the involved Web applications.
2. The user can compose access control policies using a single policy language that is supported by the user's preferred AM. As such, the user is able to easily define and apply a single policy to various resources hosted on multiple Web applications. (**R2**)
3. Security policies can be composed using a single management tool provided by AM. This allows the user to have a consistent UX when managing these policies for resources stored on different Web applications. (**R3**)
4. The user has a holistic view of the applied security controls for their Web resources and can introduce new access control policies, modify existing ones, and audit them in a single location. Moreover, access requests to resources on different hosts are evaluated centrally by AM and the user may easily audit and correlate these requests. There is no need to pull logging and audit information from distributed locations. (**R4**)

The user may choose their preferred AM based not only on its functionality but on the UX provided by this AM. Furthermore, with the proposed user-centric access control solution, different groups of users will be able to benefit from the functionality provided by a particular Web application and will be able to choose the AM which precisely satisfies their needs for access control on the Web.

4.3.5 Extensions

In the discussed UMAC approach, the user is the same entity that stores and manages resources on hosts and which defines access control policies for these resources. However, it would be possible that the user may only be concerned with managing resources and a different entity, a

Custodian, could be responsible for composing policies for these resources. A scenario has been identified where such a setting is desirable. In this scenario, young online users can use their social networking applications to interact with their peers (e.g. by sharing photos, videos, comments, or other resources). However, these users may not have the required skills or knowledge to make meaningful decisions regarding the security and privacy of their online data because of their young age (e.g. if Alice is only 13 years old she may not know if sharing her birthday photos or phone number information with random users on the Internet is considered safe). In this scenario, social networking applications would allow access control to be managed by parents or guardians of such young individuals. Access control would not be managed at social networking applications but would be delegated to a separate component. Parents or guardians could use this component to define access control policies for online data. Moreover, they could audit access requests and potentially spot any abuses. At the same time, young users could still enjoy the benefits of social networking on the Web while staying safe. The scenario is discussed in more detail in [243]. Its refined version in the context of the UMA proposal is given in [85].

The UMAC proposal allows the user to delegate access control from a set of hosts to the Authorisation Manager. In the most common configuration, a single user would have a single AM to manage access to all their distributed resources. However, the user could require multiple AMs for different hosts. For example, one AM could protect access to social data (e.g. family photos or video clips) stored on multiple hosts and a different AM could protect more sensitive information. Furthermore, different AMs could be used to protect access to different resources hosted by a single host or to protect the same set of resources on one or more hosts. A discussion about possible configurations is given in [243].

Typically, all access control decisions are issued by AM and the user is not involved in the decision making process. However, scenarios have been identified where a real-time user consent must be obtained by AM before granting authorisation to the requester [243]. The UMAC protocol does not restrict how such consent should be obtained. The AM, for example, could interact with the user over e-mail or using SMS messages. Support for real-time user consent in access control policies requires that the interaction between the requester and the AM is asynchronous. Such interaction is discussed in further chapters on the example of the UMA proposal.

Depending on the functionality of a particular AM, policies can be arbitrarily complex and may define identities and their access rights, claims, or the previously discussed real-time user consent. Third-party asserted claims, in particular, are very powerful for access control. For example, the AM may require a payment confirmation to be submitted by the requesting user. As such, the user could use a popular online gallery service to sell photos even if such service

would not support this functionality by itself. Such functionality would extend the presented claims-based authorisation [247] and this authorisation has been adopted by the User-Managed Access protocol (as discussed in Chapter 5).

4.3.6 Limitations

The refined proposal for User-Managed Access Control, despite addressing identified shortcomings as well as meeting formulated requirements (as presented in Section 1.1.2 and Section 1.1.3 respectively), still had a number of limitations. The limitations are discussed in this section.

Firstly, unlike the initial UMAC solution (recall Section 4.2), the refined version was targeted at protecting resources accessible through Web APIs (i.e. the client that would access protected resources would be a Web or mobile application and not a User-Agent such as a Web browser). If users wanted to access protected resources with existing Web browsers then such browsers would need to be equipped with plugins with support for the proposed protocol.

In order to allow interactions between different components of the proposed architecture, a certain level of user intervention would still be required. In particular, the user would need to manually register resources at the AM that would provide its access control functionality for distributed host applications. The proposed solution did not include a resource registration protocol but such functionality exists in the UMA proposal (see Chapter 5).

The refined solution was based on the access control push model where the requester application would inform the AM about the access being sought and would obtain the necessary authorisation token for this access. The requester would inform the AM about the HTTP method to be executed on a protected resource as well as the location of this resource. Therefore, the responsibility of running the authorisation phase was solely in the hands of the requester applications. Moreover, actions on resources were directly related to HTTP methods. This would limit the flexibility of the host in communicating to the AM which permissions are required for a particular access type. For example, despite the fact that a particular resource would be accessible with a single action only (irrespective of the actual HTTP method), the requester would still try to obtain authorisation for different access types (e.g. requester could consider HTTP PUT and HTTP POST as different access types even though the host would not differentiate these methods). The limitation of the push model has been addressed in the UMA solution (see Chapter 5).

Moreover, the UMAC proposal has undergone only limited testing and implementation evaluation during the conducted research. In particular, the implemented applications were basic and were not subject to substantial testing. As discussed in Section 4.1, we did not think it was useful to continue separately the work on UMAC, while concurrently UMA was being developed.

Hence, we have focused our research efforts solely on UMA, leveraging our increased understanding of issues related to user experience, design, and implementation of systems similar to UMAC. Further implementations and testing of a user-centric access control solution has been therefore done as part of our work on UMA.

4.4 Chapter Summary

This chapter introduced a novel solution to access control for distributed Web resources. It discussed the architecture of this proposal, called User-Managed Access Control (UMAC), as well as the protocol that defines interactions between different parties of this proposal. It also showed how this proposal meets identified requirements for a new user-managed access control system. It then discussed possible extensions for this proposal and provided an overview of its limitations.

The UMAC approach puts a user in full control of access to their resources on the Web. Unlike existing authorisation systems, it relies on a user's centrally located security requirements for those resources, which may be scattered across multiple distinct Web applications. These security requirements can be expressed in the form of access control policies and are stored and evaluated in a specialised user-chosen component. Applications delegate their access control function to this component.

Chapter 5

User-Managed Access

5.1 Introduction

This chapter presents the User-Managed Access (UMA) proposal [86]. UMA is a novel access management solution that consists of an architecture and a new access control delegation protocol [297; 238]. UMA provides a method for users to control third-party application access to their protected resources, residing on any number of host sites, through a centralised authorisation manager that makes access decisions based on user instructions. UMA is similar to UMAC that was discussed in Chapter 4. However, the UMA protocol differs in various ways from UMAC. Therefore, this chapter discusses the UMA proposal in detail.

The UMA proposal consists of a specialised service for authorising access to online resources and services [297; 238]. The user can define policies and impose demands on Web or mobile applications that wish to access a user's data. Importantly, such data can be spread across distributed host applications. Moreover, the user can manage relationships between online services from one location. With a specialised component being in charge of relationships, the user does not have to manually provision copies of data to potential clients, nor attempt to redefine access control policies at multiple hosting applications. Instead, requesters can be pointed to authoritative sources from which these requesters can access the data directly in a secure and efficient way.

UMA has been researched and defined by the User-Managed Access Work Group (UMA WG) [86] at Kantara Initiative [40], which includes researchers and professionals from industry and academia. The list of members of the User-Managed Access Work Group is maintained at [87]. The UMA proposal has been submitted to IETF [79]. The protocol was first proposed in [177] and [176] but has since undergone significant modifications, especially with respect to

the adoption of OAuth 2.0 [202]. This chapter is therefore based on specifications and other work published by the User-Managed Access Work Group or its participants, including [297; 84; 85; 246; 238; 274; 157; 156; 154; 247; 155; 179; 180; 178]. The UMA proposal has been evolving during the research presented in this thesis; hence, some of the discussed terminology, architectures, and protocols have changed. Such changes are not discussed in this thesis. The most recent version of the UMA proposal can be found in the publications of the UMA Work Group [86].

I have been contributing to the UMA Work Group since 2009. In particular, I have been contributing to the use cases analysis, architecture, protocol design and implementations. I have also held various leadership positions in this work group. Most importantly, since May 2010 I have been the vice-chair, use cases editor, and implementation coordinator of this work group. I have also led the research and development efforts of the SMART project at Newcastle University, which designed and implemented a complete UMA-based system.

Our shift from UMAC towards working on the UMA proposal was a natural progression. We did not think it was useful to continue separately the work on UMAC, while concurrently User-Managed Access was being developed. Hence, we have focused our research efforts solely on UMA, leveraging our increased understanding of issues related to user experience, design, and implementation of user-managed access control systems. Figure 4.1 from Chapter 4 visualises our research path.

The UMA proposal specified by the UMA WG addresses the shortcomings of existing access control solutions on the Web, which are discussed in Section 1.1.2. It also meets requirements presented in Section 1.1.3. Additional requirements used specifically during research on User-Managed Access are discussed in Section 5.2.

The remainder of the chapter is organised as follows. Analysis of additional requirements for a user-managed access control solution is provided in Section 5.2 and these requirements complement those from Section 1.1.3. Section 5.3 discusses the architecture of the UMA proposal and Section 5.4 presents the UMA protocol. An overview of the UMA trust model is given in Section 5.5. Credentials that are used in UMA are discussed in Section 5.6. Section 5.7 presents self-asserted and third-party asserted claims in UMA. Section 5.8 discusses the applicability of the UMA model to different types of information. Evaluation of UMA is given in Section 5.9 and its limitations are discussed in Section 5.10. The implementation of the discussed access control solution is presented in Chapter 6 and Chapter 7.

5.2 Requirements Analysis

This section presents requirements that were defined by the UMA WG and that were used during the research and development of the UMA proposal. Some of these requirements have been initially presented in [84]. These requirements supplement the ones discussed in Section 1.1.3. These new requirements address the shortcomings of existing access management systems such as those based on Kerberos or XACML, which can be perceived as inflexible, insufficiently user-managed and ill-fitted for the user-driven and open Web environment. Definitions of each requirement listed below are based on the definitions presented in [238] and [84].

1. **Dedicated access relationship service (R5)** Access control should be delegated from Web applications and provided as a dedicated online service. Users should be able to manage such service in order to control relationships between applications. These applications should be able to reside in distributed Web domains and they should be able to establish relationships between themselves dynamically. It should be possible to apply the functionality provided by the dedicated service for multiple different applications. Moreover, this service should not be required to understand the resources and services it protects and its functionality should be applicable to resources that can be addressed with URLs.
2. **User-driven policies (R6)** An individual user should be able to manage their own policies that would be used to make authorisation decisions for resources and services managed or owned by that user. It should be possible to apply an access control policy across distributed Web resources. Moreover, the process of composing and applying such a policy to a resource/service should be provided to the user via a consistent UX. For example, a user should be able to create a policy to share their certificates stored at one application and their "Transcript of Records" document stored at another application with the same set of users (e.g. potential employers). Moreover, access to resources/services should be given based on user-defined policies or even with the user's consent that may be given in real-time.
3. **Support for claims (R7)** Access control should not be based solely on identities because identification and authentication of client applications can be often insufficient on the Web. In such dynamic and open environments as the Web, identities of clients may not be known in advance when users compose their access control policies. Therefore, policies should allow a user to define what attributes clients should possess before access to a resource/service can be granted to those clients. Moreover, the solution should support communication between the dedicated service and the client in relation to the required

attributes (i.e. the dedicated service should be able to clearly state which attributes are required and the client should be able to provide such attributes that would be used in the policy decision making process).

4. **User-management of access control (R8)** Access relationships between services, expressed in the form of policies, should be user-managed. A user should be able to modify these relationships or even stop them completely at their own discretion. Importantly, the new solution should not require a user to be directly involved in interactions between the client applications and the applications that store resources. Instead, dedicated service and user-defined policies should dictate which interactions are allowed. The user should be able to then audit and monitor these interactions.

The UMA Work Group also included the below requirements in their research (as listed in [84] and [238]):

1. Design simplicity that would promote understanding of the solution, its implementation as well as Internet-scale deployment;
2. The use of REST principles in order to support protection of Web resources;
3. Independence from identification or authentication systems;
4. The use of OAuth for building trust relationships between parties.

A full list of requirements and design principles that were defined by the UMA WG can be found in [84].

Using OAuth as the underlying protocol for trust relationships between parties would likely support the adoption of the new solution among the increasing number of applications on the Web that already implement this protocol [188]. Therefore, UMA chose OAuth instead of a custom protocol. The use of OAuth dictates communication flows between different UMA entities.

5.3 Architecture

UMA is based on four main entities: *Authorising User*, *Authorisation Manager*, *Host*, and *Requester* (Figure 5.1). Additionally, it includes an entity, called *Requesting Party*. These entities have similar functions to UMAC entities that were presented in Section 4.3.1. The below definitions of UMA entities are based on definitions specified in [297; 238; 178]¹:

¹As discussed earlier, the UMA proposal has evolved and eventually adopted a new terminology aligned with the OAuth 2.0 protocol. The most recent version of the UMA proposal can be found in publications of the UMA Work Group [86].

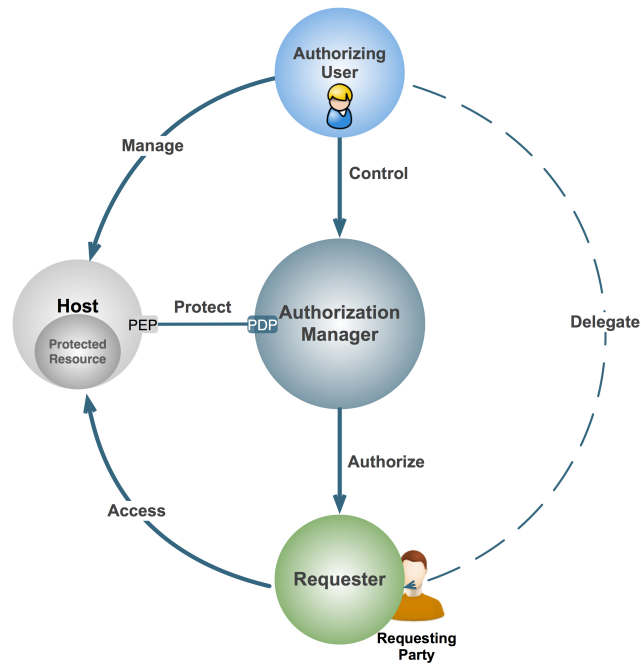


Figure 5.1: Interactions between entities involved in the UMA protocol [238].

1. **Authorising User:** An Authorising User delegates access control from their chosen Hosts to an Authorisation Manager (AM). Such a user is also responsible for configuring policies at AM and linking these policies with resources. The AM then makes access control decisions when a Requester attempts to access a Protected Resource at a Host. A user, therefore, can be considered as their own policy administrator for their resources (either resources that this user owns or merely controls).
2. **Authorisation Manager:** An Authorisation Manager (AM) acts on behalf of an Authorising User. It uses access control policies to evaluate access requests made by Requester to Protected Resources at Host application. It also issues access tokens to Requesters and these tokens are required to make authorised access requests. AM also serves Host applications and may evaluate access tokens for these applications. AM provides functionality of PAP and PDP, as defined in [302; 97] by exposing a **Protection API** for Host applications. AM also plays the conceptual role of STS as defined in [252], by exposing an **Authorisation API** for Requester applications.
3. **Host:** A Host is a Web application that is used by an Authorising User to store and manage Protected Resources. These Protected Resources (either data or services) can be shared by an Authorising User with specific Requesting Parties and Requesters. The Host delegates access control to the AM following configuration by the Authorising User. The Host is then

concerned with enforcing access control based on the permissions of a particular Requester (and this information is supplied by an Authorisation Manager). Therefore, the Host acts as PEP. Importantly, the Host can still apply its own logic to the access control decision making process.

4. **Requester:** A Requester is an application that interacts with a Host in order to get access to a Protected Resource. Access to a Protected Resource can be granted only after a Requester interacts with an AM to obtain an access token. The Requester is controlled by a person or a company (Requesting Party) that uses such an application in order to access a protected resource on their own behalf.
5. **Requesting Party:** A Requesting Party is a Web user, a corporation, or other legal person, that uses a requester application in order to get access to an UMA-protected resource. If the Requesting Party is a natural person, it may or may not be the same person as the Authorising User.

UMA focuses on protecting resources that can be accessed using third party applications and not Web browsers. Any application (e.g. mobile or Web application) that can interact with Hosts and AMs according to the UMA protocol can be a Requester. Web browsers would need to be either extended or equipped with a plugin that is able to interact with Hosts and AMs according to the UMA protocol in order for these browsers to be able to act as Requesters and access protected resources. This thesis focuses solely on applications and not Web browsers acting as Requesters.

To better understand the parties of the UMA proposal, an example scenario is given below. This scenario is similar to the one presented in [238] or discussed in Section 1.1.1 In UMA, a Web user (Authorising User) can authorise a Web application for access to a set of personal data including their CV stored at a personal data service (Host). Such access could be one-time or ongoing and this would depend on the access control policy. A user can configure a host to check with the user's preferred access control service (Authorisation Manager). The requesting party who would control the client Web application might be a recruitment company whose site is acting on behalf of the user himself to assist him in applying for one of the advertised job positions. It might also be an actual recruiter who is using a specialised application to collect resumes and match them against available job positions.

Another example for UMA is the electronic health record scenario. In this scenario, a patient might serve as an authorising user who chooses a particular AM to manage relationships between various applications. These applications may include physician, analytical laboratory, health vault, among others. These applications can act as both a host and a requesting party

seeking access to information generated by other medical hosts.

Importantly, the UMA model is applicable to various types of information. This includes information where the user is the author/source of authority, e.g. name or phone number information stored at the user's personal data service. However, the UMA model can be also applied to information where the source of authority is different from the user. This includes such information as the previously discussed "Transcript of Records" documents or various types of medical information. The applicability of the UMA model to different types of information is discussed in Section 5.8.

An extensive set of use cases with various settings of the proposed entities of the UMA solution as well as different kinds of information being protected is discussed in [85]. These use cases fit into one of the existing sharing categories that have been identified for UMA and presented in [246]:

1. Person-to-Self Sharing;
2. Person-to-Person Sharing;
3. Person-to-Service Sharing.

These sharing categories are depicted in Figure 5.2 and discussed in [178; 246]. In the *person-to-self* sharing scenario, data is shared between services on behalf of the same individual. In this scenario, the authorising user can impose specific terms (*Terms of Service (TOS)* in Figure 5.2) on the requester application. In the *person-to-person* sharing scenario, data is shared from one service with a different service used by a requesting party that is not the owner (or controller) of the data. In such scenario, the user can impose specific access constraints on this requesting party. In the *person-to-service* sharing scenario, the authorising user shares a data with a legal entity, e.g. with a specific organisation. In such cases, the TOS is imposed on the *Non-Person Entity (NPE)*. NPE is defined in [246] as "*A legal person (such as a corporation) with the capacity to take on contractual duties and obligations as a participant in an UMA interaction.*" A thorough explanation of their peculiarities, mostly from the perspective of obligations that lay on different parties in the aforementioned settings can be found in [246]. The implementation presented in Chapter 7 supports all three sharing categories.

5.4 Delegation Protocol

The User-Managed Access protocol describes interactions between all of the previously defined entities. As depicted in Figure 5.3, UMA protocol consists of the following main steps:











Requesting Party Options Authorizing User		Natural Person		Natural Person		Legal Person		
								a user, possibly a third-party beneficiary.
		TOS		TOS		TOS	TOS	option for pairwise terms of service.
		 Requester service		 Requester service				a party to access authorization
Natural Person		TOS	 Host Service					an intermediary, possibly a third-party beneficiary
	TOS	 Authorization Manager Service						
		option for pairwise terms of service	Person-to-Self Sharing	Person-to-Person Sharing	Person-to-Service Sharing	Sharing Scenario		

Figure 5.2: Sharing use cases in User-Managed Access [178].

1. User introduces host to AM;
2. User registers resources at AM;
3. User defines access control policies at AM;
4. Requester gets access token from AM;
5. Requester wields access token at host to gain access.

The aforementioned steps are defined in the UMA protocol specification in [297]. Importantly, step 3 is not a part of the UMA protocol *per se*. UMA does not impose any constraints on policy management but this step is presented in this section for the purpose of completeness.

The UMA protocol differs in various ways from the UMAC proposal, which was discussed in Chapter 4. Furthermore, our research efforts focused solely on UMA, leveraging our increased understanding of issues related to user experience, design, and implementation of user-managed access control systems. Hence, it is useful to present the entire protocol and not only its differences against UMAC in this chapter. A specification of the UMA protocol can be found in [297].

5.4.1 User introduces host to AM

In this step, a user establishes a trust relationship between a Host and an Authorisation Manager. Such trust relationship is necessary for the Host to be able to delegate its access control functionality to AM. This step of the UMA protocol consists of the following phases that are discussed separately in subsequent sections:

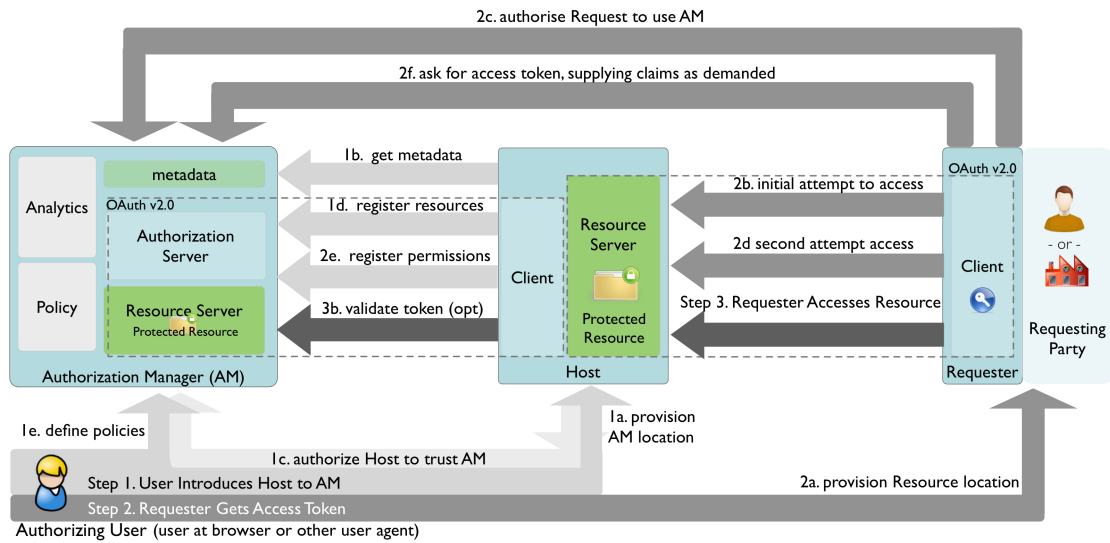


Figure 5.3: High-level overview of the User-Managed Access protocol [238].

1. Host learns the location of the user's preferred AM;
2. Host discovers AM's functionality;
3. Host registers dynamically at AM;
4. User authorises the Host for their AM.

This step of the protocol is visualised in Figure 5.4.

5.4.1.1 Host learns the location of the user's preferred AM

Firstly, the Authorising User provides the location of their preferred AM to the host application. UMA does not restrict how the Host learns such location or how the AM is selected by the user. For example, Authorising User may provide the URL of their AM to a Host by typing it into a text field on a Web page, transmitting through an information card, or by using a graphical interface [238]. This approach is based on solutions adopted by existing user-centric identity federation proposals such as OpenID [105].

Initial versions of the OpenID protocol for federated authentication required the user to provide their URL-based (or email-based) identifier during the sign in process. Such identifier was used by the application to discover the user's IDP that should be used for authentication. More recent research, such as that presented on the Google's Internet Identity Research website available at [131], shows that allowing users to use their email addresses or to select IDPs using graphical User Interfaces is a more intuitive and a less error prone approach. The recently

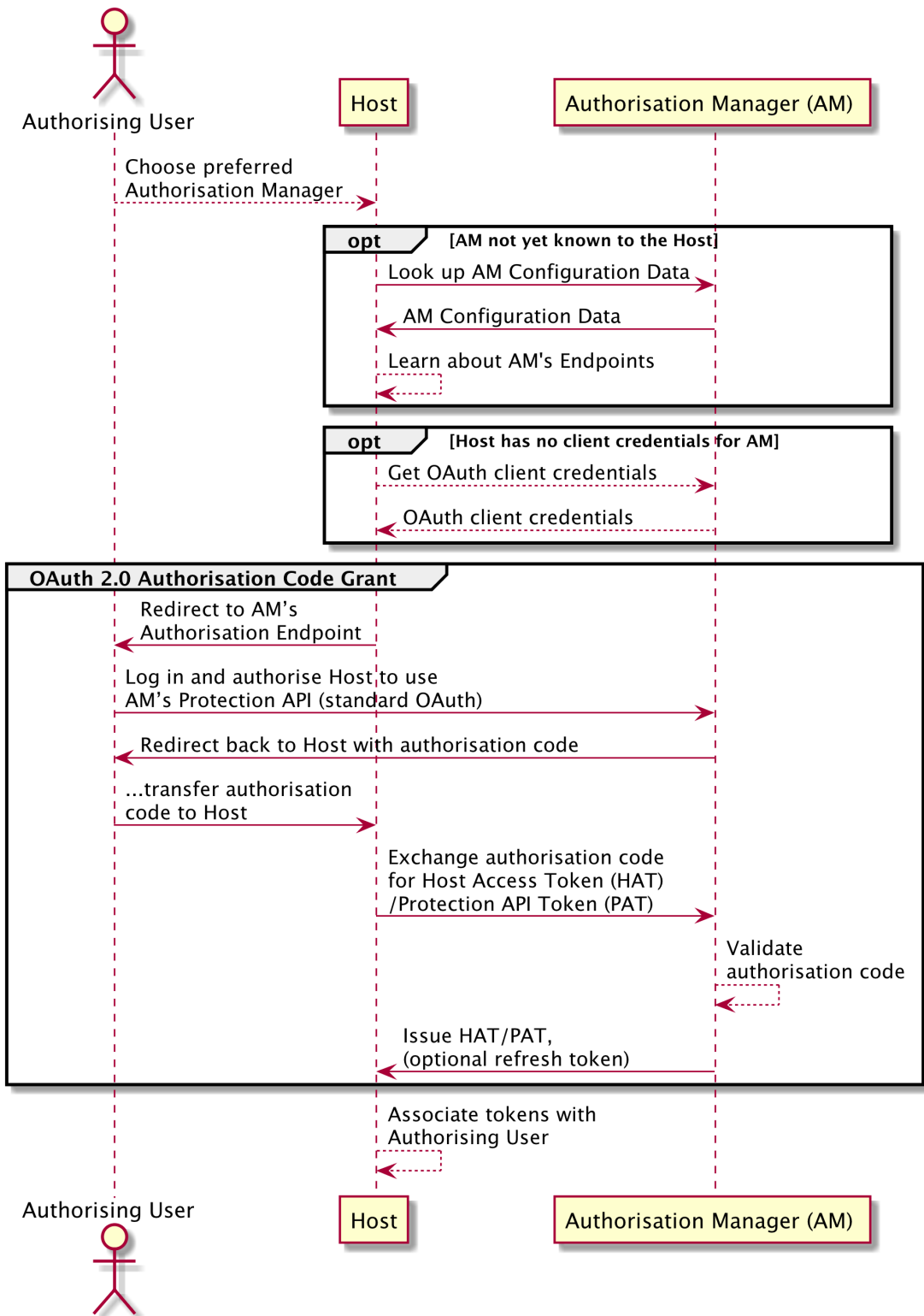


Figure 5.4: UMA Step 1: User introduces Host to AM.

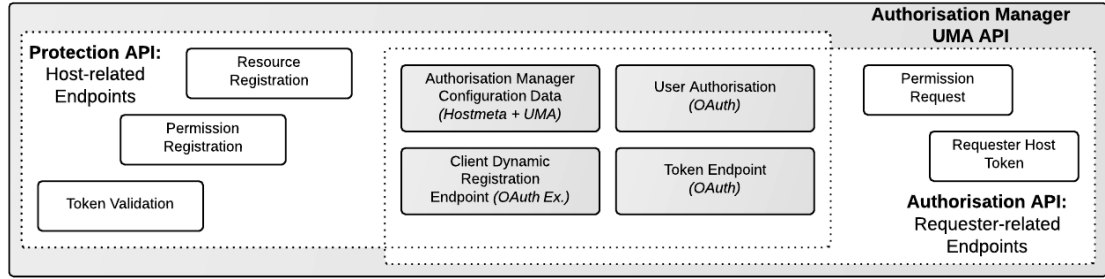


Figure 5.5: User-Managed Access API Endpoints available on the Authorisation Manager.

proposed AccountChooser [263] allows a user to securely aggregate accounts from their numerous IDPs and use these accounts across distributed Web applications which support this proposal.

5.4.1.2 Host discovers AM's functionality

When a Host is provided with the location of the AM then it uses the host-meta discovery mechanism [201] to obtain a metadata document from the AM. This document is formatted as JSON (JavaScript Object Notation) [166] and is called the *Authorisation Manager Configuration Data*. Such document resides in `/uma-configuration` directory of the AM's host-meta location (e.g. <https://www.smartam.org/.well-known/uma-configuration>).

Authorisation Manager Configuration Data document defines the location of various endpoints exposed by this AM. The defined endpoints provide functionality for Hosts and Requesters. The document also provides various configuration-related information, e.g. whether the AM supports dynamic registration for applications (which is an optional step for UMA) or what types of claims (e.g. attributes) can be provided by applications that interact with the AM (see example AM Configuration Data in Appendix G).

As shown in Figure 5.5, Host-related endpoints, called **Protection API**, include the *Resource Registration*, *Permission Registration*, and the *Token Validation* endpoints. Requester-related endpoints, called **Authorisation API**, include the *Permission Request* endpoint. The implementation presented in Chapter 7 defines an additional endpoint as part of the Authorisation API, called *Requester Host Token* endpoint². Additionally, both Hosts and Requesters can share the *Dynamic Registration*, *User Authorisation* and the *Token* endpoints (although these three endpoints can be provided separately for both application types, see Figure 5.5) [297]. These endpoints are used in further phases of the UMA protocol.

²Recent revisions of the UMA protocol have both endpoints as part of the Authorisation API.

5.4.1.3 Host registers dynamically at AM

A host application has to be registered at the AM that the user chooses for this Host. Dynamic registration allows the Host to receive the required OAuth 2.0 client credentials $\{client_id, client_secret\}$. These credentials are used by the Host when establishing a trust relationship with the AM on behalf of the user. Such trust relationship may not exist a priori and it is established solely by the user (building trust relationships is discussed in more details in Section 5.5). Importantly, credentials are used by Hosts only and are not user specific. Therefore, Hosts need to perform the registration step only once per AM, and not per user. Furthermore, Hosts can be pre-registered at specific AM and dynamic registration is only required for those AMs which are not known to host applications.

Registration of host applications is done using the *Dynamic Registration Endpoint* of the AM. This step is not defined by the UMA protocol *per se*, but relies on the OAuth 2.0 Dynamic Client Registration proposal presented in [274]. A similar proposal for dynamic registration used specifically for the OpenID Connect protocol is discussed in [264]. Dynamic registration involves providing the AM with information about the application, i.e. its name, description, URL address, redirect URL, and icon URL. The AM would display such information to the user in later stages of the UMA protocol.

5.4.1.4 User authorises the Host for their AM

During this phase, the Host obtains the user authorisation for the AM. This authorisation is required for the Host to be able to use the functionality provided by the AM, which includes registering resources that should be protected with this AM (through interactions with the *Resource Registration Endpoint*), registering permissions for Requesters (*Permission Registration Endpoint*) and validating tokens received from Requesters (*Token Validation Endpoint*).

This phase of the UMA protocol is completed using OAuth 2.0 Authorisation Code Grant (formerly Web Server flow) [202]. As discussed in [297], the Host acts in the role of an OAuth client during this flow. The Authorising User acts in the role of an OAuth end-user resource owner. The AM acts in the role of an OAuth authorisation server. During this phase, the Host completes the following two steps as part of the OAuth 2.0 flow:

1. Host obtains initial authorisation in form of a short-lived code;
2. Host exchanges the code for an access token for AM.

5.4.1.4.1 Host obtains initial authorisation in form of a code. In this phase, the Host obtains the initial user authorisation to use the functionality provided by the AM. This authorisation is in

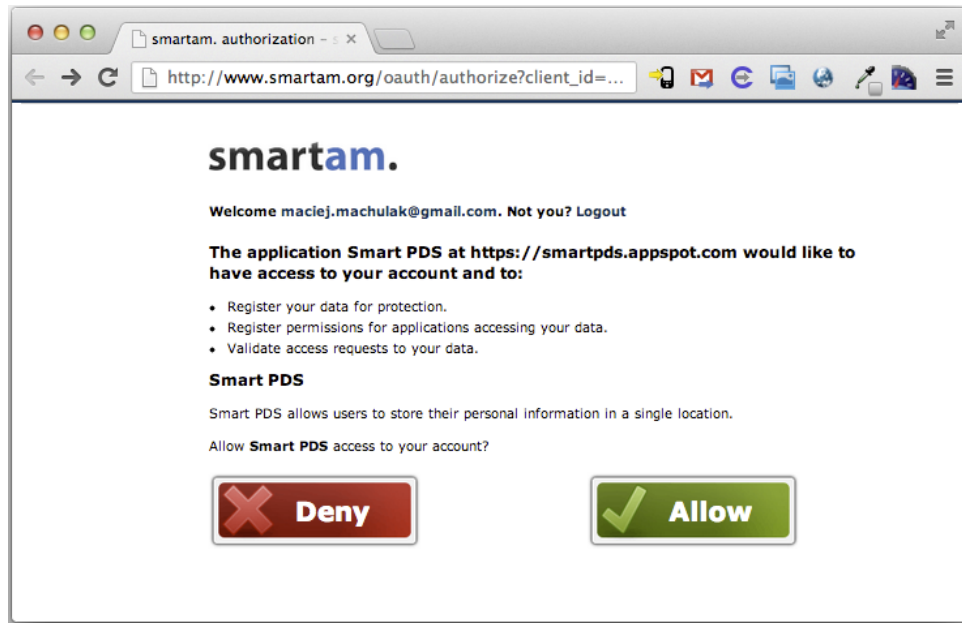


Figure 5.6: OAuth 2.0 authorisation page displayed to the User.

form of a short-lived *authorisation code* as defined by OAuth 2.0 in [202]. The host application directs the user to the *User Authorisation* endpoint at the AM, with a set of client-specific parameters (such as `client_id`, `redirect_url`, and `scope`, among others).

Before the user is presented with an OAuth 2.0 authorisation page, this user has to be authenticated at the AM. Authentication, however, is outside of the scope of UMA. Such authentication can be done using existing mechanisms (e.g. based on username and password) or with federated identity protocols, such as those proposed by OpenID, OpenID Connect or SAML.

An example authorisation page is shown in Figure 5.6. Such page can be loaded in a pop-up, with a clearly displayed URL of the AM, or in the same Web browser window during the redirect phase. The purpose of this page is to collect the user consent, which will allow the AM to issue initial authorisation to the Host. The user has to authorise the Host for a specific set of permissions, which will allow this Host to use the AM's Protection API, i.e. *Resource Registration*, *Permission Registration*, and *Token Validation* endpoints. As depicted in Figure 5.6, these permissions, and their respective OAuth scopes, can be:

1. Register your data for protection (`uma_resourcereg` OAuth scope)
2. Register permissions for applications accessing your data (`uma_permissionreg`)
3. Validate access requests to your data (`uma_validation`)

Initially, UMA would not dictate the names and granularity of OAuth scopes that need to be authorised by the user to allow the Host-AM communication. The above ones have been proposed by the SMARTAM V2 implementation, which is discussed in Chapter 7. The implemented granularity follows OAuth 2.0 best practices, where a single scope is associated with a single endpoint and access type. To simplify the process of authorisation, SMARTAM V2 has eventually used a single `uma_host` scope, which would allow the Host to obtain access to all three Protection API endpoints (more details are given in Chapter 7). More recent versions of the UMA protocol define the following scope to be used by Host applications - <http://docs.kantarainitiative.org/uma/scopes/prot.json>.

After successful authorisation, the UMA protocol follows the OAuth 2.0 flow, where a short-lived authorisation code is transferred to the Host using the front-channel communication, as a `code` parameter.

5.4.1.4.2 Host exchanges the code for an access token for AM. When the Host obtains an authorisation code, it can then exchange this code for the **Host Access Token (HAT)** at the AM's *Token Endpoint*. Further research on UMA resulted in the name of this token being changed to the **Protection API Token (PAT)**. This thesis uses the name HAT because of the implementation that is presented in Chapters 6 and 7. HAT is used by the Host when interacting with various Protection API endpoints at AM (the AM itself acts in the role of OAuth resource server by exposing these endpoints to the Host).

UMA does not define the format of HATs that are issued by AMs. Instead, it relies on the underlying OAuth 2.0 specification for this purpose. In particular, **Bearer** type access tokens are supported [210]. Such tokens are short-lived and have to be refreshed from time to time (token's lifetime is purely implementation specific). Therefore, apart from the HAT, the AM may also return a long-lived refresh token. Example token requests and responses are given in the OAuth 2.0 protocol specification. Similar requests are made by the host application to the *Token Endpoint* of AM. The software presented in Chapter 6 provides support for automatic refresh of access tokens without the need of user or developer intervention.

Once a trust relationship is established between a Host and AM, the communication between these applications is secure in terms of the integrity, confidentiality and authenticity of messages. UMA does not require the use of message level security but guarantees the aforementioned properties with transport level security (i.e. messages are transferred using HTTP over TLS/SSL [172]). Furthermore, HAT allows only a specific Host to act as the client for AM.

When the Host receives the HAT then the user can protect resources stored at this Host using the AM that issued this HAT. The Host becomes the enforcement point while the AM

acts as the PDP/PAP/PIP component. The Host also stores the $\{user_id, am_url, host_id, hat\}$ relation in its persistent store.

There can be multiple different configurations of the described access control delegation, depending on the needs of the user as well as functionality provided by the Host. In terms of user requirements, as discussed in [243], the user may decide not to "*put all eggs in one basket*" and can use multiple Authorisation Managers depending on the sensitivity of their resources, e.g. a different AM can be used to protect sensitive personal information and a different one to protect social information. From the perspective of the Host, this application can delegate access control to a single AM for all its resources, to different AMs depending on the user that manages resources, or to different AMs depending on individual resources (and irrespectively of the user). A detailed explanation of various proposals is presented in [243] and [85].

5.4.2 User registers resources for protection at AM

The user stores and manages resources on host applications as usual (i.e. the protocol does not impose any constraints on this process but only interferes with the access control functionality). The way access control is exposed to this user is application dependent but is typically achieved by providing a configuration menu that can be accessed by end users at the UI level. For example, such menu can be accessible by clicking on a security-related link associated with a resource (e.g. "*Share*", "*Sharing Settings*", "*Protect*", etc.). In UMA-enabled applications, these links may point to access control policies at AM instead of local policies (Figure 5.7).

The Host can register user's resources, in the form of *resource sets*, using the AM's *Resource Registration* endpoint [297]. Registration of resource sets is done using the resource registration protocol proposed by UMA WG and discussed in early revisions of [297]³. This protocol has been later published as an independent specification but this is not presented in this thesis. The resource registration protocol allows the Host to describe a resource set in a JSON format. The Host registers the description and manages its lifecycle at the AM through the aforementioned *Resource Registration* endpoint (see Figure 5.7).

Resource Registration endpoint is a RESTful API provided by AM as part of the *Protection API*. It supports the following operations on *Resource Sets*: *Create*, *Read*, *Update*, *Delete*, and *List*. The Host has to use its valid HAT obtained previously to gain access to this endpoint. HAT is also used to identify the Host when creating resource sets at the AM.

The Resource Registration API has a well-known structure of `{rsreguri}/resource_set/{rsid}` (e.g. `https://www.smartam.org/uma/api/resourcereg/resource_set/{rsid}`), which simplifies development of host applications. These applications only need to learn the

³The resource registration protocol has been eventually extracted from the main UMA protocol specification.

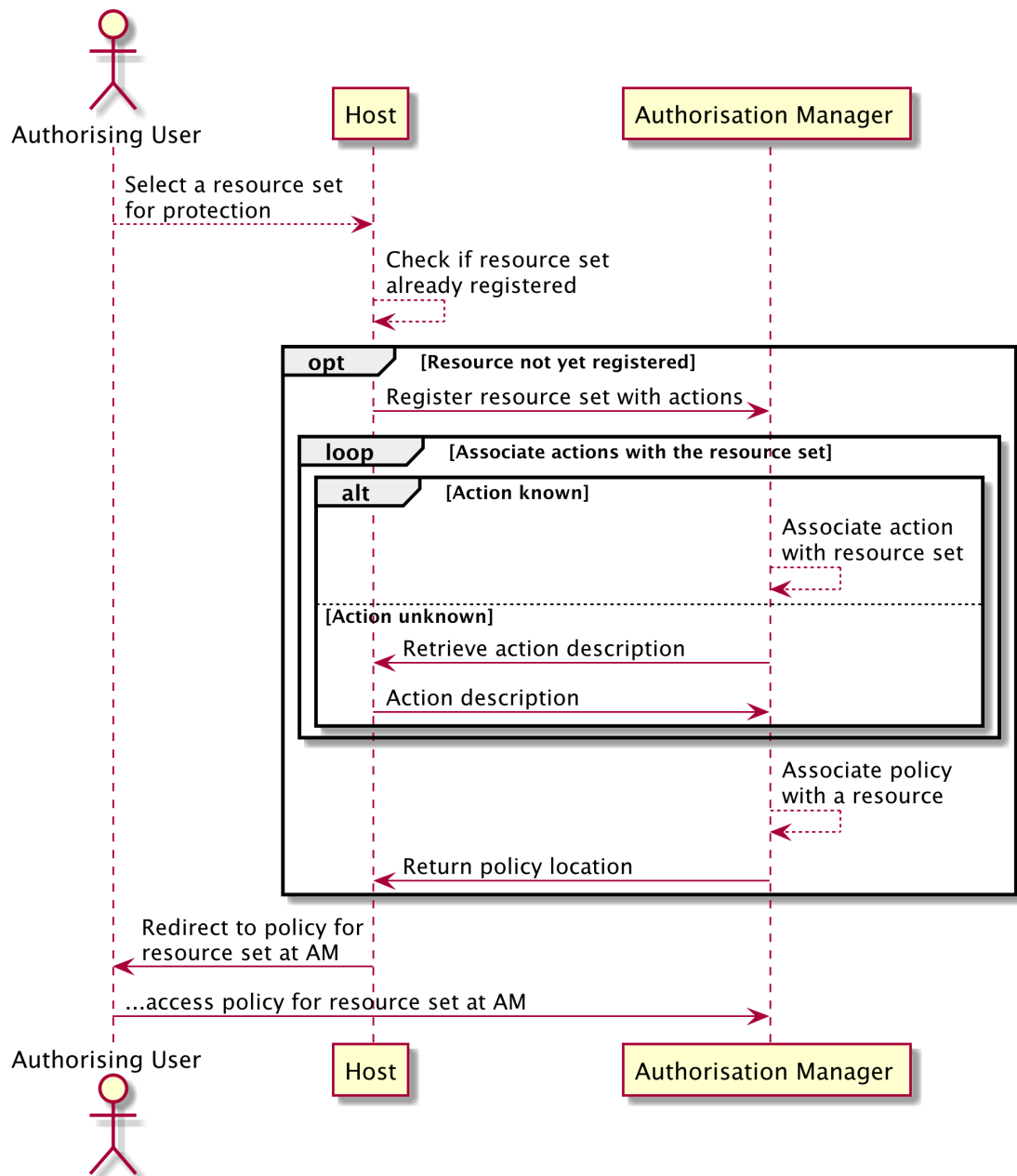


Figure 5.7: UMA Step 2: Host registers resources for protection at AM and directs a User to the policy URI at AM.

location of the `rsreguri` (through the discovery process) and are able to interact with various UMA-compliant Authorisation Managers. The `rsid` identifies a resource set and this value is set by the Host. In case the `rsid` component is not included in the `PATH` during requests to the AM, then the URI applies to the set of resource set descriptions already registered by the Host.

API operations are summarised in Table 5.1. Presented URI structure assumes a defined `{rsreguri}` `PATH` component. Presented requests and responses are formatted in JSON.

Table 5.1: Operations supported by Authorisation Manager Resource Registration endpoint.

Operation on Re- source Sets	HTTP method and URI structure	Request	Successful HTTP response	Response body of a successful re- sponse
Create	PUT /resource_set/{rsid}	Resource set description. Content type of application/uma-resource-set+json	201 Created. Location header with URI of the created resource set. ETag with the version of the registered set.	Status message, including _id, _rev, policy_uri (optional). Content type of application/uma-status+json
Read	GET /resource_set/{rsid}	No body	200 OK	Resource set description, including _id, _rev, policy_uri (optional). Content type of application/uma-resource-set+json
Update	PUT /resource_set/{rsid}	Updated resource set description. Content type of application/uma-resource-set+json. If-Match header with the version of the set being updated.	204 No Content. ETag header with the new version of the updated set.	N/A
Delete	DELETE /resource_set/{rsid}	No body. If-Match header with version of the set being deleted.	204 No Content.	N/A
List	GET /resource_set/	No body.	200 OK	The list in the form of a JSON array of {rsid} values (resource sets registered by a Host).

Table 5.2: Parameters of a resource set description.

Parameter	Description	Required
<code>_id</code>	A string that uniquely identifies the resource set. The resource set is meaningful only to the Host. However, the AM is able to map this resource set description to a particular user through the provided HAT used by the Host during interactions with the resource registration endpoint.	required
<code>name</code>	A human-readable string that describes a set of one or more resources.	required
<code>icon_uri</code>	A URI pointing to a graphical icon representing the resource set. Such icon can be displayed for the user at AM.	optional
<code>actions</code>	An array referencing one or more URIs identifying actions that this resource set supports.	required

```

1  PUT /api/uma/resource_reg/resource_set/CRrg408B HTTP/1.1
2  Authorization: Bearer 7e6e5d10e725482c34a5ac59a4b37b2b
3  Accept: application/uma-status+json
4  Content-Type: application/uma-resource-set+json
5  Host: www.smartam.org
6
7  {
8    "resource_set":{
9      "_id":"CRrg408B",
10     "name":"Phone number",
11     "icon_uri":"https://puma-pds.com/icons/phone.png",
12     "actions":["https://puma-pds.com/uma/scopes/read",
13               "https://puma-pds.com/uma/scopes/write"]
14   }
15 }
```

Listing 6: Example of a resource set registration request issued by a Host to AM.

A resource set description, as defined in [297], is an object with the name `resource_set` and with parameters as presented in Table 5.2. Example of a registration request containing a complete resource set description is given in Listing 6.

As shown in Table 5.2 and presented in Listing 6, the `actions` parameter contains list of URIs that point to descriptions of actions that can be performed on the resource set. These actions are similar to OAuth scopes, but can be applied to multiple resources (e.g. `read` action could be applied to *address* and *email* resources at a Host, while OAuth would typically differentiate between `read_address` and `read_email` scopes). It is up to the Host to define actions for different resources and make these available for Authorisation Managers in form of JSON

Table 5.3: Parameters of an action description.

Parameter	Description	Required
<code>_id</code>	A string that uniquely identifies the action at a Host.	required
<code>name</code>	A human-readable description of the scope of access for a resource set.	required
<code>icon_uri</code>	A URI pointing to a graphical icon that represents the scope of access.	optional

documents. Such actions do not necessarily need to be related to HTTP verbs. Documents that describe actions contain the `_id`, `name` and `icon_uri` parameters. These parameters are presented in Table 5.3. A document example is given in the UMA protocol specification in [297].

Both `name` and `icon_uri` parameters are used at AM to present to the Authorising User during the policy composition steps. Importantly, action descriptions can be stored anywhere on the Web and the Host is not required to store these descriptions by itself. This allows the use of standardised scope descriptions for resources (e.g a standardised set of actions could be provided for personal data stored on distributed applications). Descriptions have to be accessible to AMs through HTTP GET requests made to the URIs of action documents.

```
1  HTTP/1.1 201 Created
2  Location: https://www.smartam.org/api/uma/
3           resource_reg/resource_set/CRrg408B
4  ETag: ef21be1a-4393-44e0-bdf3-5e27c6c9ec1d
5  Content-Type: application/uma-status+json
6
7  {
8    "_id": "CRrg408B",
9    "_rev": "ef21be1a-4393-44e0-bdf3-5e27c6c9ec1d",
10   "policy_uri": "https://www.smartam.org/am/data/8/sharing_settings"
11 }
```

Listing 7: Resource set registration response containing the registered set's revision and the URI of the policy.

Successful registration of a resource set at AM is confirmed back to the Host (see example response in Listing 7). Such a response contains the revision of the registered resource and such revision is used in *update* or *delete* requests (as presented in Table 5.1) The Host can be optionally provisioned with the information regarding the policy location for this set.

Once a resource set is registered at AM, the Host must use UMA mechanisms to limit access to any resources corresponding to this resource set. The Host has to rely on the AM to supply

valid permissions for authorised access and these permissions are based on the token that the Host receives from the Requester.

It is important to note that initial versions of the UMA protocol would not define the resource registration protocol. Authorising Users would need to create their resource descriptions directly at AM and the Hosts would contact the AM whether access to resources should be granted or not. Manual resource registration is shown on the example on SMARTAM V1 implementation discussed in Chapter 7.

At the end of this step, a set of resources is successfully registered with the Authorisation Manager and the user can specify policies for these resources.

5.4.3 User defines access control policies at AM

The UMA protocol does not impose any constraints on how access control policies are composed by a user or how a policy is linked with a resource [238]. Policy management and decision-making take place solely within the AM and this information is not conveyed to other UMA entities.

The AM is meant to provide a consistent UX by allowing users to manage their policies for distributed Web resources with a coherent UI. Different AMs may provide different management tools with different user interfaces and underlying policy types or policy engines. Support for specific tools may dictate the choice of AM by a user.

UMA purposely does not constrain the policy composition process in order to support a variety of resource sharing scenarios on the Web. A user may compose policies that identify specific subjects and their access rights to a user's resources or services. A policy may also require an Authorising User's consent to be collected in real time. More importantly, the UMA protocol supports the ability of an AM to require "claims" from a Requester before authorisation is granted. The requirement for such claims can be driven by policies that are defined by Authorising Users at AM. Other examples of policies are discussed in more detail in [85].

Linking a policy to a resource may be performed in a variety of ways. A Host may offer a typical security-related user interface (e.g. the previously mentioned "*Share*", "*Sharing Settings*", or "*Protect*" link). When a user clicks on such a link then they are redirected to the configured AM to associate a resource with an existing policy or create a new policy (recall Figure 5.7). Similarly, a user may decide to log in to an AM and manually link a policy with a resource or configure an existing policy using provided management tools. The UMA implementation discussed in Chapter 6 and Chapter 7 supports both types of interactions. Section 6.3 shows a simplified process of resource registration and policy composition with a one-click solution.

At the end of this step, a set of resources is successfully associated with one or more access control policies defined by a user at the AM. The AM evaluates access requests issued by requester

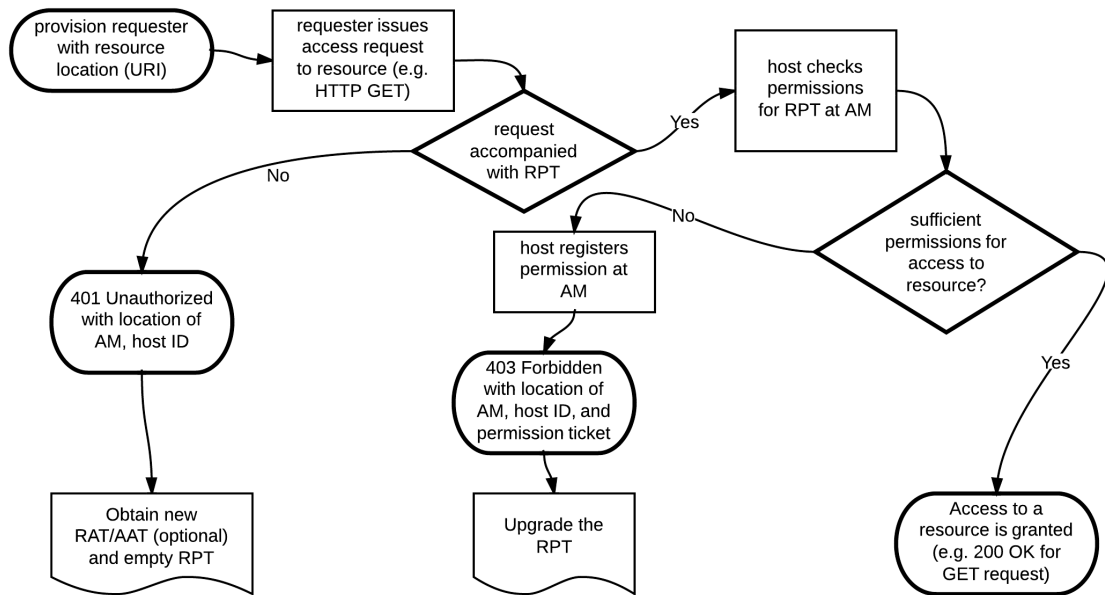


Figure 5.8: Access requests issued by Requesters can be subject to different protocol paths, depending on the presence and validity of the RPT.

applications against these policies and only then issues tokens to Requesters.

5.4.4 Requester gets access token from AM

The requester application, instructed by the Requesting Party, can access protected resources that are managed at host applications. The UMA proposal does not define how the Requester learns the location of a protected resource. Such location, in form of a Web accessible URI, can be advertised by the Authorising User for Requesting Parties. The Requester needs to know the required syntax and semantics for accessing resources (e.g. it has to understand the APIs provided by host applications).

HTTP access requests to protected resources need to be accompanied by an access token called **Requester Permission Token (RPT)**⁴. This token is necessary for the access control process to be initiated at the Host. As specified by the UMA protocol in [297], the Host can respond to the Requester's access request in one of several ways depending on the circumstances of the access request (Figure 5.8). This depends on the presence as well as validity of the RPT included in the request to the Host. Based on the response, the Requester initiates different UMA protocol flows and these flows depend on the following:

1. Access request did not contain an RPT;

⁴Requester Permission Token has been eventually renamed to Requesting Party Token. However, this thesis uses the original name.

2. Access request contained an invalid RPT;
3. Access request contained a valid RPT.

The first two flows are presented further in this section. The third flow of a Requester presenting a valid RPT is shown in Section 5.4.5. This protocol step is depicted in Figure 5.9.

5.4.4.1 Access request did not contain an RPT

Initially, if RPT is missing in the request issued by a Requester then a Host responds with a HTTP 401 `Unauthorized` response [186]. The UMA protocol uses the `WWW-Authenticate: UMA` header, to inform the Requester about the AM protecting the resource. The Host includes information about the AM (via `am_uri` parameter) as well as the Host (via the `host_id` parameter) in this header. The latter one is the identifier of the Host used at AM, i.e. the Host's `client_id`.

The Requester uses the provided `am_uri` to perform discovery of various endpoints of the AM, in case the Requester has not yet interacted with this AM. Such discovery is done similarly to the discovery used by host applications (recall Section 5.4.1.2). This discovery results in the requester application learning about the following endpoints: *Dynamic Registration*, *User Authorisation*, and *Token* endpoint. The Requester also learns about the Authorisation API, i.e. *Host Token*, and *Permission Request* endpoints.

The UMA specification at some point specified only a single endpoint named *Permission Request* endpoint as part of the Authorisation API⁵. This endpoint provided dual function of issuing new RPTs to Requesters as well as associating existing RPTs with new permissions. However, this chapter discusses two endpoints separately because of the implementation which provides support for both endpoints (see Chapter 7). *Host Token* endpoint is responsible for issuing empty RPTs to Requesters and the *Permission Request* endpoint is responsible for associating new permissions to RPTs.

The requirement for AMs to issue empty RPTs and only then associate permissions with these RPTs is related to the fact that AMs are unaware of the permissions that are required for a particular access request to succeed. Therefore, Hosts have to provide such information to the AMs and only then permissions can be associated with RPTs (after the policies are evaluated). This is discussed in more details in Section 5.4.4.1.2 and in Section 5.4.4.2.

5.4.4.1.1 Requester authorisation at AM. If the Requester is not yet registered with the AM, it registers using OAuth 2.0 Dynamic Client Registration [274]. The Requester can then proceed

⁵This was changed in more recent revisions of the UMA protocol, which now includes two separate endpoints.

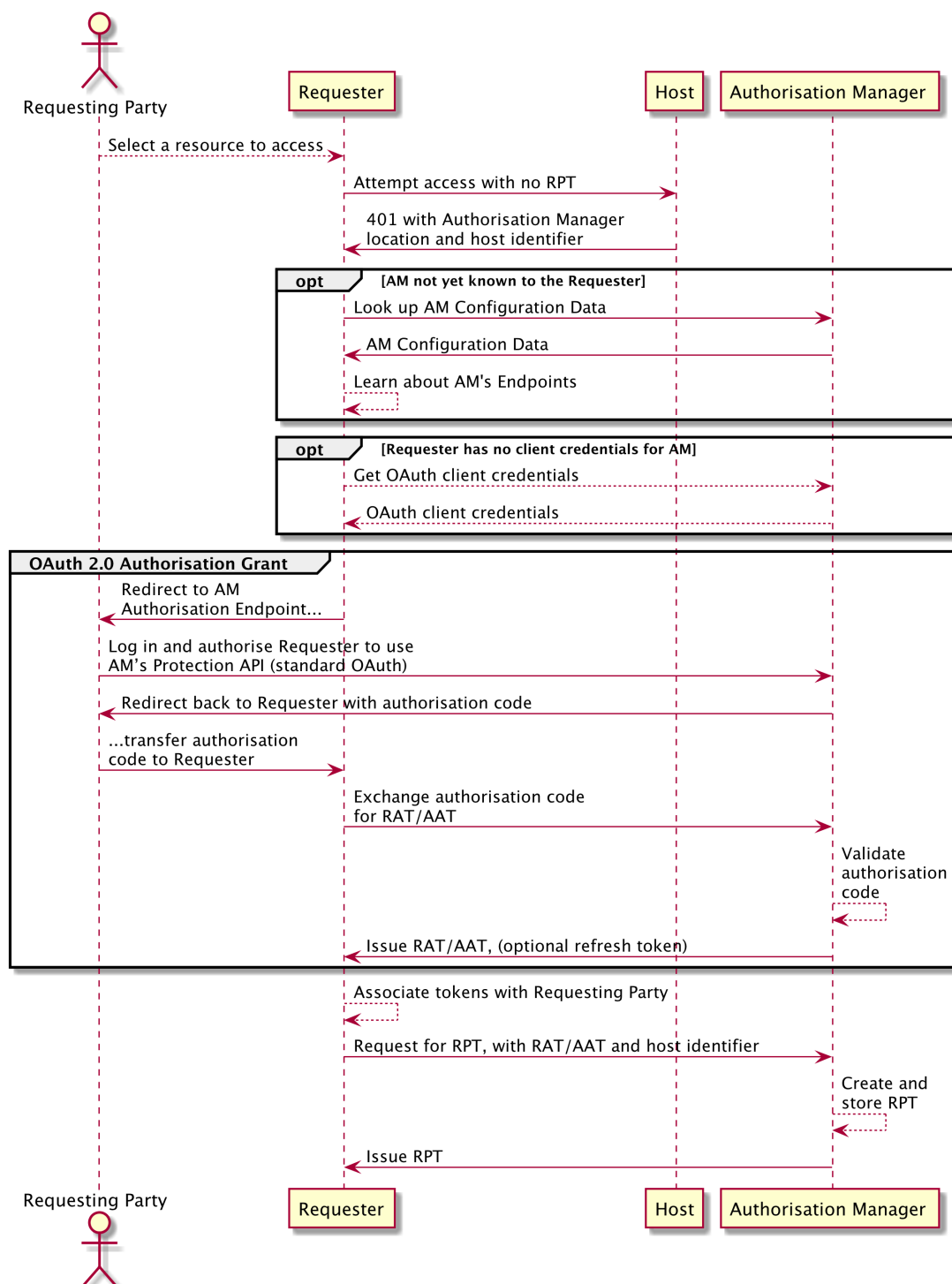


Figure 5.9: UMA Step 3: Requester gets RAT/AAT and RPT tokens from AM.

to obtaining the **Requester Access Token (RAT)**. Further research on UMA resulted in the name of this token changed to the **Authorisation API Token (AAT)**. This thesis, however, uses the name RAT because of the implementation that is discussed in Chapters 6 and 7. A part of the UMA implementation presented in Section 6.2 was based on an even earlier version of the UMA protocol with a yet different naming scheme.

Obtaining the RAT token, similarly to obtaining the HAT token (recall Section 5.4.1.4), is based on the OAuth 2.0 protocol and the Authorisation Code grant. It requires the Requester to redirect the Requesting Party to the *User Authorisation* endpoint at AM. Before an authorisation page is displayed, the Requesting Party needs to authenticate to the AM. Similarly to the UMA step defined in Section 5.4.1, this phase of the protocol does not define how the Requesting Party authenticates.

After successful authentication, the Requesting Party can be presented with an authorisation page and has to provide consent at this page to authorise the Requester to use this AM's Authorisation API (this is similar to the authorisation page presented in Figure 5.6). UMA would initially not dictate the names and granularity of scopes which need to be authorised by the user to allow the Requester-AM communication. The SMARTAM V2 implementation, which is discussed in Chapter 7, uses a single `uma_requester` scope, which allows the Requester to obtain access to all endpoints of the AM's Authorisation API. The recent versions of the UMA protocol define the following scope to be used by Requesters - <http://docs.kantarainitiative.org/uma/scopes/authorization>.

When the Requesting Party authorises the Requester at the AM, it is then redirected back to the Requester, along with a short-lived authorisation code. Such code is later exchanged for RAT. This is similar to the interaction between the Host and AM (recall Section 5.4.1.4.2).

RAT is a token which is used by a Requester to acquire access to the Authorisation API on AM. This token is usually short-lived but it can be issued by the AM to the Requester with an optional refresh token, which can be a long-lived token. Refresh token can be used by a Requester to subsequently reuse already obtained authorisation and to request fresh Requester Access Tokens from AM without the need of repeating the authorisation process at AM. The time for which both the access and refresh tokens are valid can be controlled by a user at AM or can be determined solely by an AM, and is implementation specific [238].

5.4.4.1.2 Requester obtains a new RPT for a Host at AM. When the Requester obtains the RAT, it can use the AM's *Host Token* endpoint to obtain an RPT for a host application that stores the protected resource. It can obtain such RPT by sending an HTTP POST message to this endpoint, with the identifier of the Host, and this information is used by the AM to

issue a new empty RPT. The RPT is associated with the $\{Requesting\ Party, Requester, Host\}$ tuple. Requester obtains the empty RPT once per $\{Requesting\ Party, Host\}$ tuple. Therefore, the initial request for such empty RPT can be considered negligible as far as efficiency and performance is concerned.

Initially, the RPT is not associated with any permissions. Therefore, if the Requester uses this token to access a protected resource then such access will simply fail. However, for every request that is accompanied with an RPT that is not valid, the Host registers the necessary permissions at AM and this situation is discussed in the next section.

The design choice for issuing empty RPTs is dictated by the fact that the AM does not know which permissions should be associated with an RPT for a particular access request. The AM requires this information to be provided by the Host (see Section 5.4.4.2). This allows the Host to be solely responsible for determining what are the necessary permissions for a particular access request. Importantly, new permissions can be associated with the RPT only if the relevant access control policy (or policies) is met.

5.4.4.2 Access request contained an invalid RPT

When the Requester obtains the RPT, it can use this token to access a protected resource but such access will simply fail, since no permissions have been associated with the RPT. In such case, the Host registers the necessary permissions that would be required for such access to succeed. This part of the UMA protocol is visualised in Figure 5.10.

The Host uses the *Permission Registration* endpoint at the AM. This registration requires providing the identifier of the resource set being accessed as well as the set of actions, which need to be authorised for the Requester. This information is used in further phases of policy evaluation. An example of a permission registration request is shown in Listing 8.

When the permission is successfully registered, the Requester is provisioned with the permission ticket by the host application and can interact with the *Permission Request* endpoint provided by the AM. The purpose of this endpoint is to allow the Requester to take part in the authorisation phase (recall the discussion from Chapter 2). This endpoint requires an RPT and the permission ticket to be present in the request (example of such request is given in [297]).

The UMA protocol does not define how the AM performs evaluation of access requests. For example, the AM may use simple rules or more sophisticated policy engine to evaluate access requests against applicable policies. Once the AM decides whether the access request is valid or not, it may respond to a requester application in one of three ways - a successful access response, an unsuccessful access response or a claims-requested document. The corresponding HTTP responses are listed below:

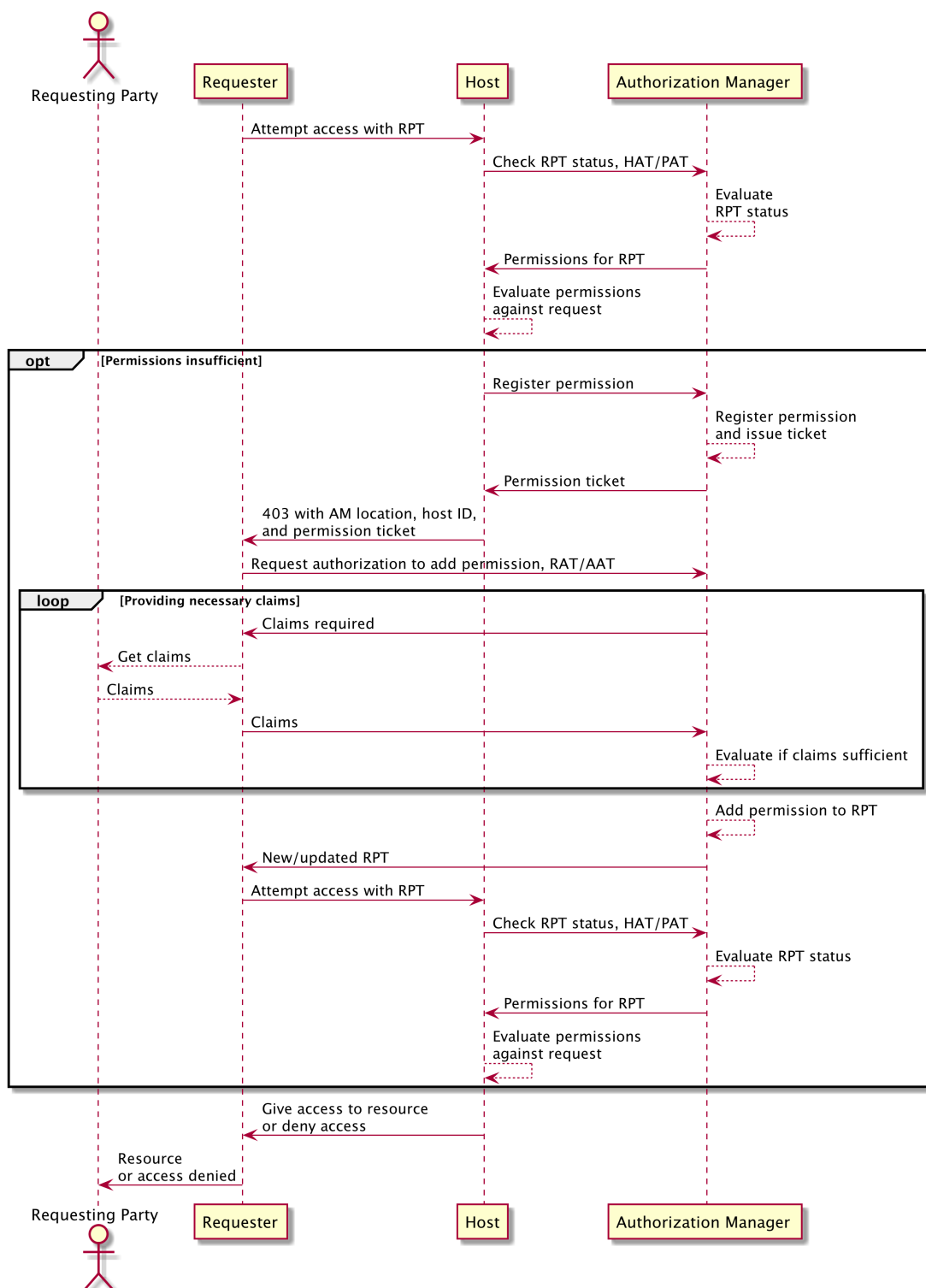


Figure 5.10: UMA Step 4: Requester wields RPT at Host to gain access to a protected resource.

```
1  POST /api/uma/permissions_reg HTTP/1.1
2  Authorization: Bearer 7e6e5d10e725482c34a5ac59a4b37b2b
3  Accept: application/uma-permission-ticket+json
4  Content-Type: application/uma-requested-permission+json
5
6  { "requested_permissions": [
7    { "resource_set_id": "6fe22d2d22ab8d59bd4fc9dd3d8c02a8",
8      "scopes": [
9        "http://static.smartam.org/puma/read.json",
10       "http://static.smartam.org/puma/write.json"
11     ] } ]
12 }
```

Listing 8: Host registers a permission at the AM's Permission Registration endpoint.

1. HTTP 401 **Unauthorized** - the Requester is unable to access a resource and it will not be able to apply for permissions;
2. HTTP 201 **Created** - response containing a newly created RPT, with associated permissions for a resource;
3. HTTP 200 **OK** - response containing the *claims-requested* document.

Firstly, the AM may decide that a Requester is definitively not authorised to access a particular resource according to a user's policy. In this case the AM responds with an unsuccessful access response and the Requester knows that it should not attempt to further access a resource.

Secondly, if the AM has all the required information concerning this particular access request then it may respond with a successful access response. For example, the issued RPT is already associated with the necessary permissions. As defined in [297], such a response contains a new or upgraded RPT with associated permissions. The Requester can proceed with this RPT to access an UMA-protected resource on a Host.

The third case is where there is a policy defined by an Authorising User and such policy has to be met before new permissions can be added to RPT. For example, a user's policy can specify requested claims that must be conveyed from a requesting party before an authorisation is granted. Such claims can be in the form of as self- or third-party-asserted identification, or even a promise to adhere to specific licensing terms.

Based on the user defined policy, the AM may engage the Requester in the authorisation phase, which is referred to in UMA as "*claims-gathering flow*". The AM responds with a claims-requested response containing a list of all requested claims. The Requester has to provide claims on behalf of the Requesting Party, which can be used by the AM in the policy decision making

process. This is discussed in more details in Section 5.7.

Based on the supplied claims and applicable policies, the AM can decide whether authorisation can be granted or not. The AM may then respond with a successful or unsuccessful access token response, or with yet another claims-requested response if more claims are needed.

At the end of this step, a Requester is in a possession of RPT issued by an AM for a specific access type to a resource on a Host.

5.4.5 Requester wields access token at Host to gain access

This step is executed when a Requester has acquired an RPT from AM and such RPT contains the necessary permissions. A Requester presents this token when attempting to access a protected resource on a Host.

A Host can evaluate the RPT locally or may use the AM for the evaluation process. In the first case, it is the Host that decides whether to grant access to a resource or not. Importantly, this decision must be based on the received access token (such token must be self-contained for the Host to allow this application to make a meaningful decision). If the token is valid then access to a resource is granted. In case the RPT is invalid then a Host registers the necessary set of permissions at AM (using *Permission Registration* endpoint) and responds with the HTTP 401 **Unauthorized** response that contains information about the location of the AM that protects this resource and the permission ticket which should be used by the Requester at AM.

In case a Host decides to use the AM for the RPT validation process, it then sends the RPT to the AM's *Token Validation* endpoint. UMA specifies this phase of the protocol as a requirement for bearer tokens, which are opaque to the Host. The request for validation is accompanied by a Host's own HAT previously acquired in step (1) of the UMA protocol (refer to Section 5.4.1). Additionally, a Host sends information regarding the resource on which access is being attempted, i.e. the resource set identifier. An example of such validation request is given in the UMA protocol specification in [297]. The AM replies with a status of the RPT token message. If valid, this status contains permissions that this RPT carries.

Remote token validation, although reducing the work that needs to be done on a Host's side, may result in a significant overhead due to the necessary communication between a Host and an AM. Therefore, the AM sets an expiration time for such status to allow the host application reuse the validation response for scalability and performance purposes. Expiration time is set using the `exp` parameter. This allows the Host to rely on the RPT token status for a specific period of time and not refer to the AM for every single access request that carries this particular RPT.

Whether decision should be cached or not is implementation specific and depends on the

Host. In particular, expiration of the token status is given in the response message (such expiration is not set using HTTP headers). Example of AM's response that contains permissions of a particular RPT is presented in Listing 9.

```
1  HTTP/1.1 200 OK
2  Content-Type: application/uma-rpt-status+json
3  Cache-Control: no-store
4  Pragma: no-cache
5
6  [ {
7    "resource_set_id": "CRrg408B0UXjJ4WIX5zYCIBe3mRb0",
8    "scopes": [
9      "https://puma-pds.com/uma/scopes/read",
10     "https://puma-pds.com/uma/scopes/write"
11   ],
12   "exp": 1300819380
13 } ]
```

Listing 9: Example of AM's RPT status response returned to the Host.

After this step of the protocol, a Requester gains authorised access to a protected resource or is denied access if the presented RPT is invalid. In the latter case, Requester can further engage with the AM and try to obtain the necessary authorisation for a resource (according to the protocol that was discussed in earlier sections of this chapter).

5.5 Trust Model

The User-Managed Access proposal involves numerous actors which include: Authorising User, Requesting Party, Host, Requester, and Authorisation Manager. UMA assumes that these actors do not necessarily have any prior relationships with each other. Moreover, these parties can reside in different domains that may be under control of different authorities. Therefore, the underlying trust model for UMA is quite complex and this model is discussed in this section. Importantly, the discussion on the UMA trust model in this section is based on existing overviews of or related to the UMA trust model that have been published in [157], [156] and [246]. The presented discussion includes the work done by Domenico Catalano.

In order to describe the trust model, it is necessary to consider relationships between the aforementioned parties of the UMA proposal. These relationships are visualised in Figure 5.11. As discussed in [157], there are three main aspects of the UMA trust model:

1. Host Registration at AM (*Host* - *AM* trust relationship);

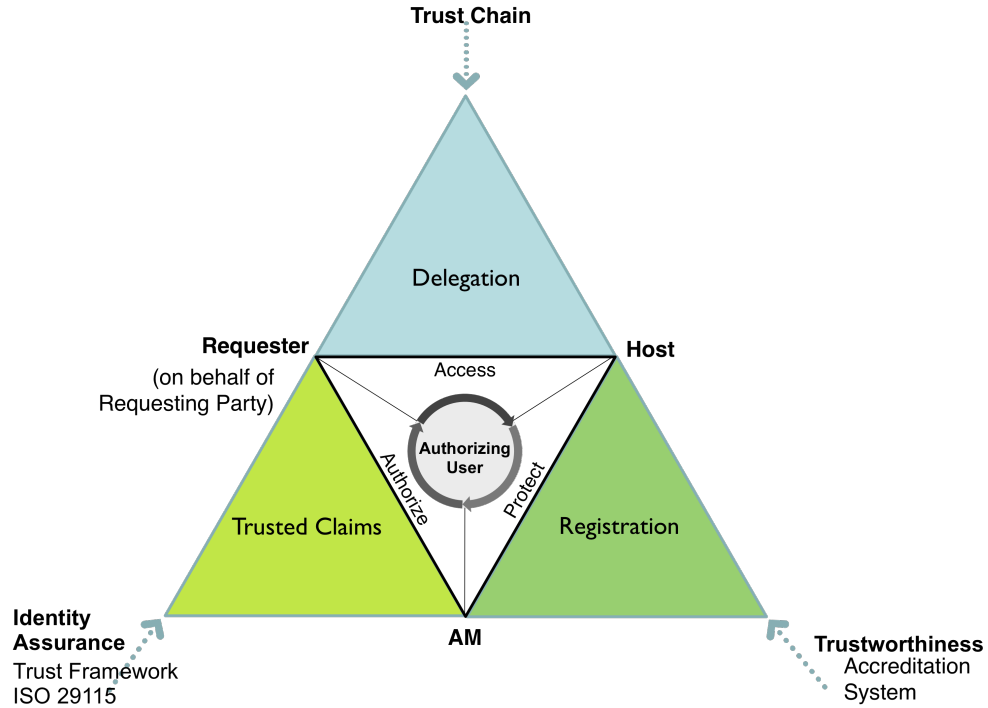


Figure 5.11: User-Managed Access Trust Model [157].

2. Trusted Claims Gathering (*AM - Requester* trust relationship);
3. Trust Delegation and Trust Chain (*Host - Requester* trust relationship).

Authors in [156] specify that the above three aspects relate to the core UMA actions, which are: *Protect*, *Authorise* and *Access*. There is also a fourth aspect of the underlying trust model for UMA, which is related to the registration of the Authorising User at the Host and AM. This aspect is referred to as *Subject Registration* [157]. Such registration is the foundation of further trust relationships and is therefore also referred to as *Bootstrapping Trust* [156; 157].

5.5.1 Subject Registration

Subject registration constitutes the initial step of bootstrapping trust in UMA. It allows to create trust relationships between the Subject (i.e. Authorising User) and the Host, as well as between the Subject and the Authorisation Manager. These two trust relationships are bi-directional.

Firstly, the user has to evaluate trustworthiness of the applications they are interacting with using quantitative (e.g. level of assurance) and qualitative (e.g. reputation) factors [156]. Secondly, applications must also trust the identity of the subject involved in the entire process (i.e. both the Host and AM have to trust the Authorising User).

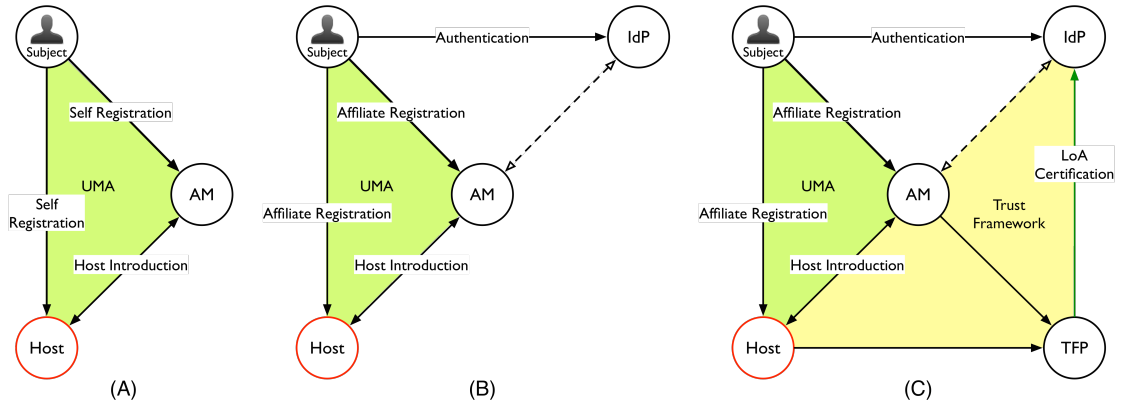


Figure 5.12: Bootstrapping Trust in User-Managed Access [156].

[156] defines at least three different approaches to bootstrapping trust (see Figure 5.12). These approaches are:

1. Self Registration;
2. Affiliate Registration;
3. Trust Framework-based Registration.

Self Registration (Figure 5.12 (A)) is the simplest approach of bootstrapping trust and requires the user to manually establish a relationship with an application (either Host or AM). The user registers for an account at these applications as usual (e.g. by filling in a registration form). The user is solely responsible for establishing their identity at these applications. Because it is the user who provides information directly to these applications and such information does not need to be trusted by these applications then Self Registration is also referred to as *Untrusted Registration* [156].

In Affiliate Registration (Figure 5.12 (B)), the user registers with their Host and AM applications using existing identity federation technologies, such as SAML or OpenID Connect. In this approach of bootstrapping trust, Host and AM applications have to trust the Identity Provider that acts as a source of trustworthy information about the user.

It is important to note that the IDP has to trust that information (e.g. user attributes) is released to trustworthy host or AM applications. This trust is based on the consent that the user provides at the IDP. The Authorising User bootstraps the trust with the IDP using self registration, affiliate registration or trust framework-based registration. Bootstrapping trust with the IDP, however, is outside of the scope of this thesis.

Trust Framework-based Registration (Figure 5.12 (C)) is similar to Affiliate Registration. In this type of registration, the Identity Provider is the source of trustworthy information about

the Authorising User. However, in this approach, the Host and AM applications may achieve a higher level of assurance in user identity as there may be a requirement for the Identity Provider to be in the same Trust Framework as the Host and AM applications. In such setting, it is possible to achieve business, legal and technical trust⁶, which can be monitored by the Trust Framework Provider (TFP) [156]. For example, before a particular application is included within a trust framework, this application may need to be contractually bound to specific business or legal terms and conditions.

5.5.2 Host Registration at AM

Host registration is the first phase of the UMA protocol. During this phase, the Authorising User delegates access control from a Host to AM. It is the user who explicitly informs the Host that access control should be delegated to an external service. Other configurations can be also present in the UMA proposal (e.g. preconfigured relationships between host applications and AMs) but are not discussed in this section.

Before access control is delegated from a Host to an external service, the user has to have relationships with these two applications. The user also has to trust these two applications and this was discussed in the earlier section. Delegation of access control from the Host requires the following two steps to be completed:

1. Host is registered with AM;
2. Authorising User authorises Host at AM.

Host registration at the AM can be done either in advance or can be dynamic (recall Section 5.4.1.3). Such registration establishes a bi-directional relationship between the Host and AM. In particular, the AM issues credentials to the Host and then trusts that any requests related to the UMA protocol come from this particular Host to which such credentials were issued. Similarly, the Host can communicate with the AM and trusts this AM to provide a specific functionality (i.e. access control related functionality as defined by the UMA protocol).

When the Authorising User delegates access control from Host to AM, this is done by establishing a trust relationship between these two applications that will communicate with each other on behalf of this user. This relationship is established by the Authorising User who is involved in the following tasks:

1. Verifying that the correct Host application will be given authorisation for the AM;

⁶Trust relationships between computing systems and services require trust to exist at each of these three levels. Firstly, relationships between services can be constrained by contractual agreements that define what services can or cannot do. Secondly, there have to be legal enforceability of these agreements. Lastly, necessary technical measures must exist to support business and legal trust between services.

2. Verifying that access control will be delegated from the Host to the correct AM.

Therefore, it is the Authorising User that establishes a bi-directional trust relationship between the Host and the AM. As discussed in Section 5.4.1.4.1, it is important that the user is aware of the Host application from which authorisation is delegated as well as the AM which will be responsible for protecting resources or services on this Host. Ensuring that the user can verify trustworthiness of both host and AM applications is based on the trust that the user has initially bootstrapped with these applications (refer to Section 5.5.1). There are various ways of supporting users with verification that they are indeed interacting with the correct application but these require some form of user involvement. Reliance on these ways is considered one of the limitations of the UMA proposal and it is discussed in Section 5.10.

On the open Web, host applications may possibly interact with various AMs. However, in certain scenarios and for certain use cases, such as financial or health-care, host applications may only be allowed to interact with Authorisation Managers which meet specific requirements at the business, legal, and technical levels. Authors in [157] introduce the notion of an accreditation system into the UMA model. Such system can provide the necessary foundation for increasing the level of trustworthiness between the Host and AM applications. An Authorising User may only be allowed to introduce host applications to trusted Authorisation Managers. A detailed explanation of the use of such accreditation system in UMA is presented in [157].

5.5.3 Trusted Claims Gathering

In UMA, the AM is likely not to have any prior relationship with the Requester that acts on behalf of a Requesting Party (this is true when the Requester does not know the AM). Such relationship is necessary for the AM to be able to evaluate whether a particular Requester satisfies a specific access control policy and whether this Requester should be given authorisation to access a resource/service on behalf of a Requesting Party. This requires that the AM can trust the information that is provided by the Requester for the policy evaluation process.

On the other hand, the Requester has to trust that the AM will use the provided information only for the purpose of policy evaluation and not for anything else. For example, the Requester has to trust that by providing some attributes to the AM these attributes will not be passed by the AM to other applications to impersonate the Requester.

Therefore, there is a requirement for a bi-directional trust relationship between the Requester and the AM. This relationship is initiated when the Requester first interacts with the Host and learns from this application the location of the AM that protects a resource. The Requester then interacts with the AM to obtain authorisation for that resource.

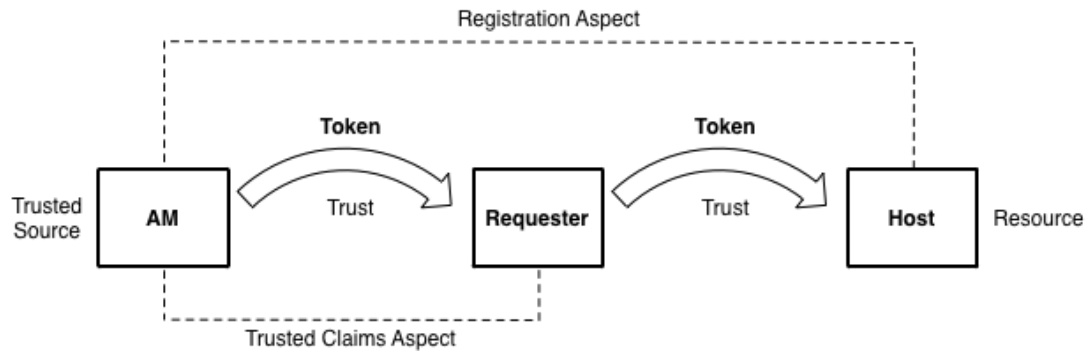


Figure 5.13: Trust Delegation and Trust Chain in User-Managed Access [157].

The AM must ensure that the information provided by the Requester is trustworthy before using such information in the policy decision making process. Therefore, the AM may refer to trusted third party Identity Providers or Attribute Providers to obtain this information. In UMA, such information provided by Requester applications is referred to as claims. Examples of claims that can be obtained by the AM include the user's name, date of birth, or phone number information, among many others.

As discussed in more detail in Section 5.7, claims must be issued by Trusted Third Parties (TTPs) and must be verifiable by AMs in order to be considered trustworthy. Therefore, the trust model in UMA for the Requester-AM relationship relies on the trust framework that allows the AM to trust claims issued by TTPs [157].

5.5.4 Trust Delegation and Trust Chain

The trust relationship between the Requester and the Host application is built on two concepts: *Trust Delegation* and *Trust Chain*. Both concepts for UMA have been introduced in [157].

Trust delegation is related to the fact that the AM issues authorisations (RPT tokens) to Requesters and such authorisations are later used at the Host. The token, representing permissions, is delivered from the AM to the Requester and subsequently from the Requester to the Host (i.e. the trust is delegated from the AM acting as the source of this trust to the Host).

Trust chain refers to previously established trust relationships, which include the Host-AM and Requester-AM relationships that were discussed in earlier sections. These relationships allow Host and Requester applications to determine the AM as the source of trust. Trust delegation and trust chain are visualised in Figure 5.13.

The AM uses the trusted claims to evaluate policies during its authorisation process. Based on the outcome, the AM may issue an RPT to the Requester. The Requester uses this RPT

when interacting with the Host. Before access to a resource can be granted, the Host leverages its trust relationship with the AM to check whether the RPT can be trusted (i.e. the Host checks that the RPT has been issued by a trusted AM that protects this particular resource/service on this Host) and what permissions this token carries.

Importantly, the trust relationship between the Host and the Requester is bi-directional and does not only involve the Host trusting the Requester through the chain of trust (as discussed earlier). The Requester also has to trust the Host that it will not abuse it. For example, if the Requester obtained the RPT based on a submitted payment that would satisfy a policy for a resource, then the Requester trusts the Host that it will provide access to this resource.

5.6 Credentials

There are multiple tokens involved in the UMA proposal and these tokens represent credentials used by different UMA entities. This section summarises the tokens and presents their relationships with existing UMA actors (see Table 5.4).

Firstly, the Host Access Token / Protection API Token represents credentials, which bind the Authorising User, the Host and the Authorisation Manager. These credentials allow the Host to use the AM's Protection API. A new HAT/PAT must be obtained for each Authorising User at a Host. It also uniquely identifies this user at AM (e.g. when the Host registers resources and uses a particular HAT/PAT then these resources are registered for a specific user).

Secondly, the Requester Access Token / Authorisation API Token, binds the Requesting Party, the Requester, and the Authorisation Manager. This token, similarly to HAT/PAT, must be obtained for each RP and Requester pair separately.

The Requester Permission Token links the Requesting Party, the Requester, the Authorisation Manager and the Host application. In particular, RPT represents the credentials of a Requesting Party, using a particular Requester, when accessing a resource (or resources) stored at Host. RPT contains specific permissions of Requesting Party and Requester pair for resources of Authorising Users at Host, i.e. it can provide access to multiple different resources.

5.7 Claims

The UMA protocol does not constrain users in composing access control policies for their resources (subject to AM implementation limitations). Rather, it allows the AM to inform the Requester about the necessary claims that should be submitted and that will be used in the

Table 5.4: Relationships between tokens and actors of the UMA proposal.

Actor	HAT/PAT	RAT/AAT	RPT
Authorising User	✓		
Host	✓		✓
Requester		✓	✓
Requesting Party		✓	✓
Authorisation Manager	✓	✓	✓

access control decision making process. This allows users to specify very flexible policies which are not only based on identities of users accessing data but also on arbitrary attributes of these users. As discussed in [154], the claim can be defined as *"An assertion made by a Claimant of the value or values of one or more Identity Attributes of a Digital Subject, typically an assertion which is disputed or in doubt."* [4].

There are different types of claims that can be used in UMA and these were presented in [238]. Firstly, claims that are submitted by requester applications may be affirmative, representing a statement of fact. An affirmative claim can be asserted by a Requesting Party or another claims issuer (e.g. such a claim can be signed by a third-party service). A statement of fact might be *"The requesting party is over 18 years of age."* or *"The requesting party is the owner of the bob@gmail.com email address."*

Secondly, a claim can be also promissory and can be asserted by the Requesting Party specifically to the Authorising User. For example, such claim may state that *"The requesting party will adhere to the specific Creative Commons licensing terms indicated by the AM in accessing this resource."* or that *"The requesting party will use the resource only for job application related purposes."* As discussed in [238] and [246], it is envisaged that the process of demanding and submitting claims would have legal enforceability consequences as necessary. For example, submitting a claim can be recorded by the AM, also for accountability and audit purposes.

The UMA protocol does require that any specific types of claims are used. Therefore, a Requester has a certain level of flexibility how it provides such claims to AM [238]. The protocol can potentially make use of already established types such as these proposed in [181; 151; 111; 103], and [115]. However, some of these types may be too complex and therefore not well-suited for Web 2.0, and all these types appear to lack the ability to specify required parameter values in claims. Therefore, these types may be unable to satisfy complex use cases such as these discussed in [85].

UMA would initially provide support for a new simple and extensible claim type based on the Claims 2.0 specification [247]. More recently, an OpenID Connect Claim Profile has been defined for UMA as well. Further information on the use of claims is given in the UMA protocol

specification in [297].

5.7.1 Claims 2.0

Claims 2.0, as published in [247], allows to express both self-asserted claims and third-party asserted claims in a lightweight way using a widely supported JSON format [166]. This format allows for a variety of claim types to be requested and submitted which increases the flexibility for users to demand the exact properties which are required for the access control decision making process. For example, the policy can demand that Requester provides the age of a Requesting Party or asserts that the age is above a certain pre-defined threshold.

Claims 2.0 allows to express claims that are requested and claims that are submitted. The first type is called the *claims-requested document*. This document contains a list of all required claims that must be conveyed by a Requester on behalf of the Requesting Party. The latter one is a list of all claims that are submitted by Requester to AM. An example structure of both document types is given in the Claims 2.0 specification in [247]. A derivative implementation of this proposal is discussed in Chapter 7 on the example of a prototype AM. Example of a claims-requested document is also given in that chapter.

5.7.2 Trusted Claims

UMA supports third-party asserted claims. This section provides an overview of such claims and the presented discussion is based on the work published in [154].

Third-party asserted claims can be explained on a person-to-person data sharing scenario presented in [154]. The Authorising User wants to restrict sharing of a resource to a specific Requesting Party (based on its identity) but such identity is not established at the AM. Instead, this identity would be based on supplied attributes from the Requesting Party. For example, a resource at a host application can be shared with a user who carries an attribute that they are the owner of the *bob@gmail.com* email address (i.e. the Requesting Party can submit a claim that they indeed own this email address).

Importantly, the AM has to trust the third-party identity claim issuer to consider the submitted claim trustworthy and to be able to use it in the decision making process. The AM may also need to make sure that only specific issuers can assert specific claims. For example, not every issuer should be able to provide information that the Requesting Party is the owner of the *bob@gmail.com* address. This is the role of the trust model that was discussed in Section 5.5.

By providing support for third-party asserted claims, UMA allows for Claims-Based Access Control (CBAC) (recall Section 2.3.1). In CBAC, the decision to grant access to a protected

resource is made using information about the subject (i.e. its attributes) when such information can be considered trustworthy. UMA WG has identified a set of use cases for such access control, including enterprise-type scenarios and social Web type scenarios (these scenarios are given in [85]).

The Authorising User can use CBAC to restrict access to their protected resources based on attributes of Requesting Parties. These attributes must be issued by Identity Providers or Attribute Providers that are trusted by Authorisation Managers. These IDPs or APs are collectively referred to as Trusted Third Parties (TTPs) [154]. When the AM requests claims it acts as a Claims Requester and an IDP or AP that issues these claims acts as a Claims Provider. Importantly, there has to be some form of a trust framework to allow the AM consider the claims trustworthy (recall Section 5.5).

5.7.2.1 OpenID Connect Claims

In UMA, OpenID Connect allows third-party asserted claims from distributed sources to be collected and transmitted to AMs. Such claims can then used for the access control decision making process, as discussed in [157].

The UMA protocol provides an OpenID Connect claim profile [58], where the AM acts as the OpenID Connect client when interacting with OIDC-compliant IDPs. Claims are obtained from the *UserInfo* endpoint that returns claims about the authenticated user. Which claims should be accessed by the AM depends on the scope of authorisation requested by this AM and such scope depends on the actual access control policy applicable for a resource/service. For example, if the Authorising User defines a policy that a protected resource can be shared with someone who is the owner of a particular email address then AM has to be authorised by the Requesting Party for a scope which will allow access to such verified email address (this is the **OIDC email** scope).

OpenID Connect specification [58] defines the following claim-related scopes for which a client can be authorised: **profile**, **email**, **address**, and **phone**. The profile scope allows the AM to access end-user's name, gender, date of birth, among others. The **email**, **address**, and **phone** scopes are self-descriptive. The UMA implementation presented in Chapter 7, allows the use of the **email** scope. Support for additional claims is planned in the near future.

In OpenID Connect, claims are associated with the issuer. The AM can check if such issuer matches one of the trusted ones (e.g. the AM checks if the identifier matches one of the pre-registered identifiers). In OpenID Connect, the issuer identifier is globally unique - it is a case sensitive URL using the HTTPS scheme that contains scheme, host, and optionally, port number and path components of the OIDC IDP. Moreover, OpenID Connect uses JWS [38]

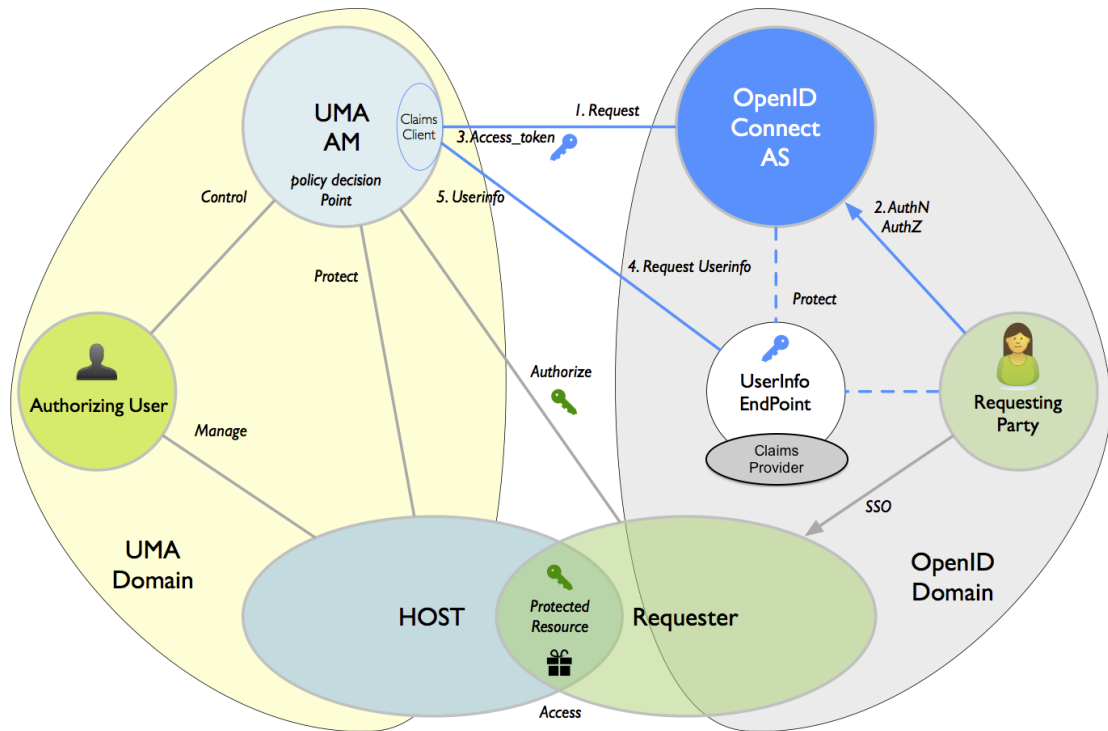


Figure 5.14: UMA conceptual model with Trusted Claims [155].

for signing claims and this allows the AM to verify authenticity of the claims using PKI. More information on trusted claims and the use of OpenID Connect in UMA is given in [157]. A basic implementation of the OpenID Connect profile is discussed in Chapter 7.

5.8 Protected Information

The UMA model is applicable to various types of information. This includes information where the user is the author/source of authority, e.g. name or phone number information stored at the user's personal data service. However, the UMA model can be also applied to information where the source of authority is different from the user. This includes such information as the "Transcript of Records" documents or medical information.

Resources, which are not owned by the user, are typically stored on systems managed by other entities and the user is merely given some control over access to this data. For example, the student does not own their "Transcript of Records" document but may be given the ability to share it from the University's institution. Similarly, the patient may not own the data that is stored at an online health vault (e.g. it may be the hospital that is the owner of this data) but this user merely provides consent for others to access their medical information.

When the user is both the owner and controller of the data then it is at the sole discretion

of this user how such data is shared and accessed by other users and services. For example, the user can provide *"read"* or *"read and write"* type access to some of their resources for other users and services on the Web. Importantly, there are no specific constraints that would be imposed on such permissions being granted by the user for others. In the UMA model, the AM can issue tokens to requester applications based on evaluation of user-defined policies. The Host can act accordingly based on the permissions which are returned by AM for a given client (i.e. permissions associated with the RPT token determine the type of access that will be given for the client).

However, when the user selectively shares resources which are sourced from others, e.g. organisations, then certain restrictions may apply. In such situations, organisation-wide or application-wide policies may still need to be combined with user-defined policies and can even take precedence over how resources are shared and accessed (depending on the actual policy combining algorithms). In such case, the decision making process in UMA would not only take place in the AM and this process may also need to take place within other components (e.g. additional decision points) or within the Host itself. Furthermore, such process may take place at different stages of the UMA protocol, either when the RPT is being issued to the Requester or when the actual access request happens.

For example, the user could use their AM to grant permissions to a Requester to access their academic related data stored within their Higher Education institution. However, such data would still only be shared according to specific University regulations. Such regulations could be managed in the form of access control policies stored at some central PDP/PAP that would be integrated with various hosting applications at the HE institution. Therefore, not every permission granted at the AM may allow the Requester to successfully access the user's data. For example, an application could be given *"read"* access to the *"Transcript of Records"* document that is hosted at a University's personal data service. However, this application may still not get access to the resource if it does not meet specific criteria set by the University. This criteria may include, among others, membership of the requester application in a particular trust framework or adherence of the Requesting Party to specific additional licensing terms.

As discussed in Section 5.4.3, the UMA protocol does not impose any constraints on how policies are specified or linked with resources that should be protected. Therefore, UMA is very flexible in this regard and can be combined with various mechanisms for policy management. The user can define their policies in one language at AM and other potential policies could be defined elsewhere and in a different language.

When the access control decision making process involves multiple policies then the policy management and policy evaluation may become challenging. Firstly, policies do not necessarily

need to be managed in single location (such as AM) but are more likely to be spread across different access control components. For example, policies can be managed independently at different PAPs that are integrated with numerous and potentially incompatible PDPs. Moreover, multiple different access control rules have to be considered before access is granted or denied.

Policies from multiple sources could be used by the AM when issuing RPTs during the "claims-gathering" flow. On the other hand, additional policies that should apply to a resource or a set of resources could be used by the Host (or any of its decision making components) when evaluating access requests which already have RPT tokens in place. Because the role of the AM component in UMA is to address the requirements of individual users and not combine multiple policies from different sources then it is likely that the latter situation would occur and the AM itself would not participate in managing or evaluating multiple policies. The Host that stores the resource may not know in advance which AM will be used for management of user policies and therefore it would be the Host that would be concerned with any additional policies. In this setting, the AM would evaluate the policy (or policies) specified by end-users before issuing RPT tokens to Requesters and any further decision making would be outside of UMA.

There are various challenges that need to be considered in architectures where multiple policies are applicable to a particular resource or a set of resources (some of these challenges were discussed in Section 2.6). Firstly, because different policies have to be combined, there needs to be a way to easily evaluate them and calculate effective permissions for requester applications. In the UMA model, policies themselves are not exposed outside of the AM to the Host. Instead, the AM provides RPT validation functionality informing host applications about permissions associated with RPT tokens. The Host (either itself or with the use of any decision points) would need to combine this information with any additional policies in place (e.g. organisation-wide or application-wide policy). Therefore, there must be a mechanism that would translate the permissions, currently encoded in the JSON format, to a format that can be easily combined with any other applicable policies (e.g. XACML-encoded policies).

When evaluating multiple policies, it may be the case that access control rules are contradicting or inconsistencies occur (recall the discussion in Section 2.6.1.4). Therefore, policy conflict resolution protocols have to be adopted as well. Combining policies and resolving any potential conflicts could be done at the Host side or at any of access control components integrated with the Host and this is outside of the UMA model. Importantly, conflicts would need to be resolved at runtime and not by using static conflict resolution mechanisms (unless of course there would be a way to get access to all policies that apply to a specific resource).

Because policies are evaluated in multiple locations, then the user has little knowledge about the effective permissions which are associated with their resources. In particular, the user can

only see and verify permissions which are set directly at the AM. The UMA model does not define any interface or protocol that would allow such AM to pull any policy information from host applications or any third party access control policy components. Such information could be presented to the user who could verify if the resource is shared as required. Without such information, the user does not have the necessary knowledge regarding the security settings that apply to their data and loses the ability to easily audit such security.

Moreover, if any additional access control checks are done by the Host (either directly or when the Host interacts with any additional decision making components) then the user has little knowledge about how resources have been actually shared and accessed. For example, the Requester may be able to negotiate access to a particular resource (by submitting the necessary claims to the AM) and then issue an access request to the hosting application. Even though the permissions may be sufficient to access a resource (according to the UMA model), the Host may still decide not to provide access to a resource or a service. For example, a particular Requester may not meet specific criteria set at the Host by the organisation that manages this application. In such case, the trust between the Requester and the Host is also broken (such trust relationship is discussed in Section 5.5.4).

When multiple rules need to be checked or when policies are evaluated in separate locations before access to a resource can be granted then performance becomes one of the challenges as well. This relates to the performance of policy evaluation when different and possibly contradicting rules have to be evaluated by a policy engine or a set of engines. Performance also relates to the number of messages that have to be exchanged between different components of the access control solution.

The aforementioned challenges related to protecting resources that have sources of authority different from the user who manages those resources are often addressed in other access control proposals such as XACML or VOMS, among others. XACML was discussed in Section 3.2 and VOMS was discussed in Section 3.10. These proposals rely on centrally located security requirements which can be defined by administrators or end-users. Both proposals do not give users the ability to dynamically establish relationships between hosting applications and authorisation components. Instead, such relationships are predefined (and often established during deployment time and not at runtime). With well-defined authorisation components, the aforementioned challenges, which relate to policy management and evaluation, can be addressed. Firstly, policies are evaluated in a single component (or a set of well-defined components). These policies are written in the same language (e.g. XACML) which simplifies their evaluation. Any potential conflicts can be detected using static analysis and not necessarily at runtime. Moreover, if authorisation is done within a single component then the performance can be improved because

less messages need to be exchanged between the hosting application and the authorisation component. However, these proposals lack the necessary characteristics required by a user-managed access control solution. We discussed those characteristics in Section 1.1.3 and Section 5.2.

5.9 Evaluation

The proposed User-Managed Access architecture and protocol address all identified shortcomings discussed in Section 1.1.2 by meeting all of the requirements described in Section 1.1.3 and Section 5.2. Evaluation of the UMA solution was initially published in [238].

The inclusion of an Authorisation Manager within the UMA architecture and allowing users to delegate access control from host applications to this component satisfy requirements **R1** and **R5**. A central component responsible for access control may provide its functionality in order to support complex Web transactions, similar to the one presented in Section 1.1.1. Requirement **R5** is further satisfied by the fact that all the entities of the proposed architecture can reside in distinct Web domains and interactions between different parties (namely Hosts, Requesters and AM) are based on the standard HTTP protocol and follow the REST architectural style [187]. The AM does not need to understand the identifiers used by various hosting Web applications. Therefore, UMA can be deployed in "today's Web". Additionally, the AM does not impose any constraints on what resources may be protected and its functionality can be applied to arbitrary Web resources or services.

An Authorising User can be involved in all the steps of access control policy management such as those given in [97] and discussed in Section 2.6.2.3. Importantly, the user can serve as their own policy administrator in UMA and it is the user that applies access control policies to their Web resources or services and grants authorisations to Requesters of these resources using the AM. The UMA proposal does not constrain how the user defines access control policies and the user is free to choose their preferred AM that offers tools that meet user's requirements and expectations (**R1**). Having a single AM, user can be provided with a consistent UX when composing policies and linking these policies with distributed Web resources or services. Hence, the UMA proposal also meets requirements **R2**, **R3** and **R6**.

The UMA protocol supports communication of claims between Requesters and the AM and this was discussed in Section 5.7. Such claims are used by the AM in the authorisation process and allow a user to define policies that define arbitrary attributes of requesting parties that may need to be requested before authorisation can be granted. Policies do not have to be based solely on specific identities of potential requesting parties but rather on attributes. Therefore such policies fit well into the highly dynamic and open Web environment where servers may not

know in advance possible identities of clients accessing data. In UMA, policies may also impose contract terms that govern ongoing access to data and may require requesting parties to convey their agreement to these terms. UMA relies on legal means to enforce such agreements (more information on this topic is given in [246]). The aforementioned properties of the UMA proposal meet requirement **R7**.

With access control being delegated from various Web applications to a user's preferred AM, a user may easily manage relationships between their online services [238]. Access control decisions for distributed Web resources and services are performed by this central component. A user does not have to be directly involved in interactions between services accessing data and services hosting data as these interactions happen according to pre-defined policies. These UMA properties satisfy requirements **R4** and **R8**.

As discussed in [238], the UMA solution is fundamentally different from existing proposals for access control that exist both within closed and open environments. Unlike classic access management systems such as Kerberos [255] and XACML [97], UMA allows all interacting entities to establish relationships dynamically. Moreover, it allows the user to play a pivotal role in managing policies for their distributed Web resources and to serve as their own policy administrator. The UMA model, however, can be used in parallel with such technologies as XACML or Kerberos and this was discussed briefly in Section 5.8. Moreover, unlike protocols such as OAuth 1.0/2.0 [200; 202], the UMA solution allows to aggregate the authorisation for many point-to-point relationships between applications. Such relationships have been rapidly emerging on the Web and the UMA provides means to secure them.

5.10 Limitations

The User-Managed Access proposal, despite addressing identified shortcomings as well as meeting formulated requirements, still has a number of limitations.

Firstly, host and requester applications have to be registered at the AM in order to interact with this AM. Without registration, a malicious application could try and obtain an authorisation for AM. Firstly, the user would be given no information about the identity of the application they would be authorising at the AM. The user would need to rely on the redirect URI to understand where they will be redirected after authorisation and such manual check can be insufficient. Similarly to problems discussed in [59], malicious applications could direct the user to AM and could try to use either open redirect servers or by exploiting XSS flaws [7] on trusted applications in order to get authorisation for the AM. Therefore, the user must be provided with a clear explanation of the application that will be given the authorisation. Existing OAuth

providers, which allow unregistered applications to use the OAuth flow for their APIs (through the use of *anonymous credentials*), show a very cautious message to the user on the authorisation page.

Application registration itself does not necessarily solve all the problems related to the trust that the user should have when establishing a relationship between an application and AM. During dynamic registration, applications can potentially provide any information that they want to the AM. Therefore, the AM must not trust this information and must make necessary steps to inform the user about this at the authorisation page. For example, the AM could inform the user that the application is registered dynamically, provide the time of such registration or even state if the registration has been verified (e.g. manually by the administrator) at the AM.

In UMA, the human being is the sole point of establishing a trust between the Host or Requester and the AM. Requiring the user to check application's identity or its redirect URIs is one of the limitations of UMA. The AM itself can support users in this process, e.g. by validating if registered application names are sufficiently different from existing and reputable ones or that redirect URIs do not contain query parameters (which may mean that an attacker is trying to exploit an open redirect on a reputable site). The AM may limit the format of such redirect URIs thus giving the user better information on the actual location that they will be redirected to after authorisation.

As discussed in [225], user-centric systems, such as the UMA proposal, are prone to phishing attacks that may trick users to reveal confidential information (e.g. username and password) to malicious third parties during redirects. For example, user can be redirected to malicious AM where they can be tricked to provide their credentials for the real AM. Existing research, such as that discussed in [131], proposes that the address bar with the URL is clearly displayed to the user and that the redirect happens in a standalone Web browser window (and not within an `iframe`). As discussed in [287; 279; 301; 170; 303], unfortunately, existing Web browser security indicators may not work efficiently for users. Indicators that can be installed at Web browsers as plugins [205] are also subject to these problems.

Host and requester applications obtain authorisation using the OAuth 2.0 protocol, which relies on authorisation code being transferred using front channel communication. This authorisation code can be possibly exposed to the browser itself or other malicious third parties. Furthermore, any messages contained in the query of the URL are normally stored in log files of HTTP servers as well as any HTTP proxies through which access to the Host/Requester or AM takes place. Unfortunately, this problem occurs irrespectively of whether HTTP is used over TLS/SSL or not. Therefore, it is important that only a short-lived and one-time use authorisation codes are transferred through the user's Web browser.

When the user is redirected to AM to authorise a host application, this user often started the OAuth 2.0 flow by themselves. However, in case of requester applications, the Requesting Party might have started the flow by trying to access a protected resource and not necessarily knowing anything about the AM that protects this resource. Therefore, the Requesting Party, when authorising a Requester at the AM, should be presented with clear information on why they have been redirected to the AM while trying to access a protected resource. UMA currently does not discuss the necessity of such information to be shown to the Requesting Party, which is one of the usability limitations of the UMA proposal. Such information could be presented either at the AM or preferably at the Requester before redirecting the Requesting Party to this AM.

UMA is targeted at protecting resources accessible through Web APIs (i.e. the client that would access protected resources would be a Web or a mobile application and not a User-Agent such as a Web browser). If users wanted to access protected resources with existing Web browsers then such browsers would need to be equipped with plugins with support for the UMA protocol. This is one of the limitations of the proposal as users would need to install additional software in such cases.

The AM is the sole point of issuing authorisations to Requesters and validating tokens for host applications. Therefore, the AM is single point of failure. If such component is either compromised or is malicious then the data that is managed at hosting applications can be exposed. In particular, in the current version of the protocol there is nothing that would allow the Host to further require requesting applications to authenticate themselves at the Host. This differs from the Street Identity protocol, where the Relying Party has to authenticate at the Attribute Provider and must include an authorisation token before access to a verified attribute can be granted. This limitation of the UMA proposal could be addressed by using secure and possibly various AMs. As discussed in Section 5.4.1.4.2, the user may decide not to *"put all eggs in one basket"* and can use multiple AMs depending on the sensitivity of their resources, e.g. a different AM can be used to protect online documents and a different one can be used to protect online photos.

Having the AM in the centre of interactions between different components imposes certain constraints on scalability of the entire solution. In particular, if the AM becomes unavailable and cannot issue authorisations or validate tokens then access to resources cannot be granted even if the Requester can communicate with host applications. This limitation of UMA can be addressed by not imposing constraints on authorisations that are issued to requester applications. Such authorisations can be self-contained and issued for a specific period of time and may be validated by Hosts without the need of remote token validation at the AM. For example, the

authorisation can be encrypted with a secret that is shared between the AM and the Host and this authorisation may contain all the information needed to make an access control decision at the Host.

Moreover, UMA allows to centralise access control for distributed applications that can manage various types of resources and services. Therefore, UMA provides a significant benefit to users who no longer have to manage their access control settings in different locations but can use the provided UX of their preferred AM for access management. Having a single component responsible for security to various resources requires that the provided UX and functionality needs to be adapted to various types of resources. For example, the user experience at the AM needs to take into account that this AM may be used for protecting financial information as well as personal data. Therefore, it is unlikely that such UX will be optimal for all resource types and this may be considered as one of the limitations of UMA. Centralising access control significantly outweighs the described shortcoming and the users gain a significant benefit of improved control over their distributed data with a centralised solution such as AM.

Similarly to existing IDPs, the AM has a view on applications that are used by the user. However, the AM also knows about relationships between these applications. Therefore, if such AM is compromised then not only the attacker can get access to resources of the user but can also get an insight into existing transactions of that user. Moreover, the attacker can get insight into relationships of a particular user with specific requesting parties (i.e. other users or institutions). It is therefore essential to provide the necessary security for the AM.

The Requester needs to refer to the AM in order to obtain the authorisation for resources at the Host. In particular, it is the Host that decides on the scope of access for which the authorisation should be given (recall permission registration described in Section 5.4.4.2). Therefore, the Requester may need to refer to the AM numerous times in order to build the necessary set of permissions for a single resource (e.g. read, write, manage) or for multiple resources. This requires numerous messages to be exchanged between the AM and the Requester and affects the performance of the protocol. UMA does not contain a mechanism which would allow the Requester to communicate to the Host using a single request that multiple permissions may be required. This limitation of the UMA proposal, however, clearly separates the roles allowing the Requester to have little knowledge about the internals of the host application. Therefore, the Requester is only concerned with referring to the AM once access at the Host cannot be granted immediately.

UMA is currently tightly bound to OAuth 2.0 and this may be perceived as one of its limitations. Therefore, its communication flow follows the one proposed by OAuth 2.0 (where the client forwards credentials to the AM and obtains a capability that is later used to interact

with the Host). However, using OAuth as the underlying protocol for trust relationships between parties would likely support the adoption of the new solution among the increasing number of applications on the Web that already implement this protocol [188].

5.11 Chapter Summary

This chapter presented the User-Managed Access solution. It started with presenting additional requirements for UMA. Then, the chapter discussed the UMA architecture and its protocol. It also discussed credentials used in this proposal and presented the UMA trust model. It also showed applicability of UMA to different types of information. This chapter also contained the evaluation of UMA and its limitations.

UMA provides a method for users to control third-party access to their protected resources, residing on any number of host sites, through a centralised authorisation manager that makes access decisions based on user instructions. UMA gives users the required flexibility in sharing their data and supports requesters with accessing such protected data, similarly to the previously discussed UMAC proposal.

Chapter 6

Host and Requester Frameworks

6.1 Introduction

This chapter presents User-Managed Access frameworks¹ that allow applications to externalise their access control functionality and act as Hosts and Requesters, as defined by the UMA protocol. Developed frameworks additionally contain specific enhancements to simplify the process of interactions between applications of the UMA proposal.

In particular, this chapter presents two distinct frameworks that have been designed and implemented for different programming and deployment environments:

1. UMA/j - framework for applications written in Java (Section 6.2);
2. Puma - framework for applications written in Python (Section 6.3).

UMA/j and Puma frameworks were developed as part of the JISC/HEFCE-funded SMART project [77]. Some parts of Section 6.2 that discusses UMA/j have been first published in [239], as a joint effort with Łukasz Moreń. Parts of Section 6.3 that discusses Puma were first published as blog posts in [123] and [124], as a joint effort with Jacek Szpot.

UMA/j and Puma were designed and implemented during different times of development of the emerging UMA proposal. This chapter restrains from providing revisions of the protocol implemented by this software. This is due to the fact that both frameworks were continuously improved over time. Development of both frameworks allowed to provide valuable feedback to the UMA WG and such feedback was often used during work on the core UMA protocol specification. The differences between both frameworks are summarised in Table 6.1.

¹This thesis uses the terms "framework" and "library" interchangeably, despite the fact that these two terms are sometimes used to describe slightly different types of software.

Table 6.1: Comparison of UMA/j and Puma frameworks.

UMA Actor Support	UMA/j	Puma
Host	Yes	Yes
Requester	Yes	Yes
Authorisation Manager	Yes	No
Functionality		
Integration with SMARTAM 1	Yes	No
Integration with SMARTAM 2	Yes (early versions only - no permission registration)	Yes
Resource Registration	Yes (with SMARTAM 2)	Yes
Permission Registration	No	Yes
Claims 2.0	Only self-asserted claims	Self- & third-party asserted claims (both types through redirects)
Token Naming		
Host-AM Token	Host Access Token (HAT)	Host Access Token (HAT)
Requester-AM Token	Requester Access Token (RAT)	Requester Access Token (RAT)
Requester-Host Token	Authorisation Token (AT)	Requester Permission Token (RPT)
Implementation		
Programming language	Java	Python
Programming paradigm	Imperative	Functional
Deployment	Web application container	Google App Engine
Tests	Unit/integration tests	Unit tests
Configuration	XML-based	In Python code

UMA/j supports development of UMA Authorisation Managers, Hosts and Requesters. Parts of this framework were used to develop SMARTAM V2 that is discussed in Chapter 7. Therefore, this chapter only discusses the host and requester part of UMA/j. These parts are called UMA/j Host and UMA/j Requester.

Initial versions of UMA/j Host and UMA/j Requester were used to build client applications that were integrated with SMARTAM V1. The framework was then used for integration of these client applications with early versions of SMARTAM V2. However, such integration was eventually abandoned with the introduction of *permission registration* and *permission tickets* to the UMA proposal and to the SMARTAM V2 implementation. These two concepts are not supported by UMA/j Host or UMA/j Requester. Therefore, UMA/j is discussed from the perspective of its integration with late versions of SMARTAM V1 and some parts (e.g. RAT tokens) are shown from the perspective of their integration with early versions of SMARTAM V2. UMA/j also implements a modified version of the Claims 2.0 specification [247]. The claims module allows Requesters to submit self-asserted claims, similarly to those discussed in Section 5.7.1.

In terms of implementation, UMA/j is written in Java and provides a high-level *UMA API* for applications. It also allows to use low-level APIs for finer control of UMA protocol flows. It was implemented during early stages of research on UMA. Therefore, its integration with Web applications still required a significant level of knowledge of the underlying protocol, which can be considered a major issue with this software. UMA/j follows an object-oriented design with different software modules responsible for different steps of the UMA protocol. UMA/j comes with unit and integration tests in place. It can be used in Java Web applications, which conform to the Java Servlet specification [101]. It provides additional support for the popular Spring framework [74] and its configuration can be provided in an XML configuration file.

Puma, on the other hand, was developed as a more lightweight framework and it provides support for building host and requester applications only. This framework was used to build client applications for a more recent version of SMARTAM V2. Furthermore, Puma provides support for permission registration and permission tickets. This framework supports only basic Claims 2.0 proposal through redirects of Requesting Parties to an AM. This support allows for both self-asserted as well as third-party asserted claims to be used at an AM (see Chapter 7).

Puma is built primarily for applications running on the Google App Engine Platform as a Service. Puma is written in the Python language. Because this framework was designed and implemented later than UMA/j, it contains various enhancements and simplifications for application developers. In terms of programming paradigms, this framework follows a functional design [208; 295]. At the time of writing, the framework only contained basic unit tests but

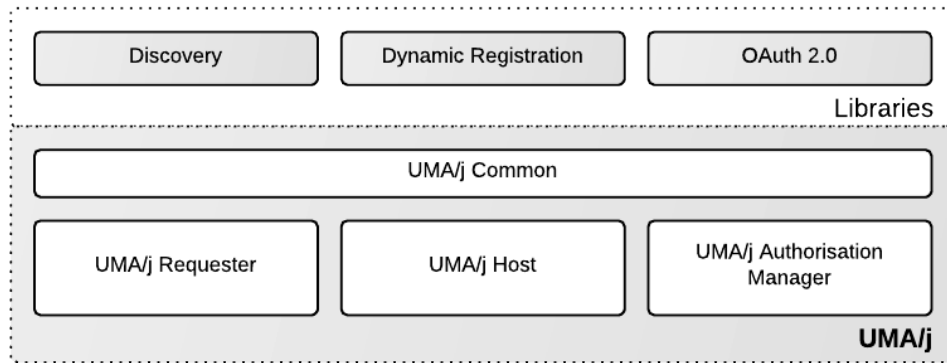


Figure 6.1: High-level overview of the modules in the UMA/j framework and related libraries.

integration tests are planned to be provided in the future. Unlike UMA/j, this framework requires configuration to be provided directly in the Python code.

6.2 UMA/j

UMA/j has been built as the first known framework to provide Web applications with support for the UMA protocol. An overview of the host part of this framework was published in [239] as a joint effort with Łukasz Moreń and this section of this thesis is based on that publication. A high-level overview of modules comprising the developed software is shown in Figure 6.1.

UMA/j can be integrated with new and existing Web applications to allow these applications to become UMA-compliant Hosts capable of delegating access control to users' chosen Authorisation Managers (*UMA/j Host*). It also provides the mechanisms for applications to become UMA-compliant requesters that can access Web resources protected with the UMA protocol (*UMA/j Requester*). Furthermore, it also provides a module for building UMA-compliant Authorisation Managers and this module has been used when building SMARTAM V2 (*UMA/j Authorisation Manager*). Importantly, UMA/j uses additional libraries that were developed as part of the SMART project, namely *Discovery*, *Dynamic Registration*, and *OAuth 2.0*.

UMA/j abstracts away specifics of the underlying UMA protocol and allows applications to use high-level abstractions to comply with the protocol. By providing a single *UMA API*, the framework allows developers to keep a clean separation between the business logic of their applications and the proposed security mechanism. In terms of programming paradigms, UMA/j follows an imperative design [295], which puts emphasis on changes in state.

The framework implements one of the early revisions of the UMA protocol. It has been initially developed to allow integration of client applications with SMARTAM V1. It was later adapted to allow integration with a more complex but more functional SMARTAM V2. However,

this second integration was done only with an early version of SMARTAM V2 when the UMA proposal did not have permission registration in place. Further implemented client applications that were integrated with SMARTAM V2 leveraged the Puma framework that is discussed in Section 6.3.

The remainder of this section is structured as following. First, this section discusses the Common module and related libraries that are used by Host, Requester, and AM parts of the UMA/j framework. It then presents the Host module followed by the Requester module. This section does not discuss the UMA/j AM module. Instead, the AM implementation that uses UMA/j AM is presented in Chapter 7. This section also discusses the limitations of the UMA/j framework. UMA/j implementation and evaluation are discussed in Appendix C.

6.2.1 UMA/j Common

The UMA/j Common module provides shared code, which is used by Host, Requester, and AM parts of the framework. A description of each package of this module is given below:

uma.common.domain.*

POJO classes used among different modules of the UMA/j framework, e.g. used to store AM information, access and refresh tokens, requests and responses, error messages, etc.

uma.common.exception.*

Implementations of different exceptions, including checked and runtime exceptions.

uma.common.providers.*

UMA data provider interfaces, used by host and requester applications. Common in-memory implementations are provided for developer convenience.

uma.common.validator.*

Validators used for requests and responses used in the UMA protocol.

uma.common.utils.*

Various utility methods related to token handling and exception-to-error mappings.

These packages are used in different parts of the UMA/j framework. These classes have been combined in a single module to ease development of UMA/j AM, UMA/j Hosts, and UMA/j Requester. In particular, having implementations of messages and validators in a single module allowed to ensure the consistency among different parts of UMA/j.

6.2.2 Discovery

The discovery module implements the Well-Known Location [260] and Web Host Metadata [201] discovery. It allows the client application to obtain information about services available at a user's chosen AM. Such information is provided in a JRD document [201] that describes the locations of endpoints exposed by AM and the AM's supported access token formats, among other information. Acquired information from the configuration document is later consumed by the Core module and is available to other UMA/j components. Discovery is an optional step in UMA and can be omitted by applications that have been pre-registered in any other way (e.g. manually by the application developer).

Initially, discovery would constitute an integral part of UMA/j. However, since it implemented a protocol that was independent from the UMA protocol itself, it has been provided as a separate library. Therefore, it can be used by other libraries than UMA/j. Furthermore, this module was originally developed to consume XRD documents [116] which were used in SMARTAM V1. This module was adapted to support more recent revisions of the UMA protocol and to allow processing of JRD documents (see example JRD document in Appendix G).

6.2.3 Dynamic Registration

The Dynamic Registration module allows host and requester applications to receive the required OAuth 2.0 client credentials $\{client_id, client_secret\}$ that must be used when establishing a trust relationship with an AM on behalf of a user. Dynamic registration involves providing an AM with information about the application that an AM displays to a user during the authorization step.

Dynamic registration is performed according to one of the models defined in [237] (with further changes in [274]) where an application pushes information to an AM (*Push* model) or where an AM discovers information directly from an application (*Pull* model). UMA/j extends the Pull model by allowing an AM to respond with a nonce, which a client must place into its descriptor file (e.g. XRD or JRD file). An AM then retrieves this file from the client's discovery URL and verifies its validity. In both models, similarly to requirements for the OAuth 2.0 protocol, a transport level security must be provided with the use of the TLS/SSL.

In the Pull model, an AM has more flexibility in determining the trustworthiness of a Web application because a description of such application is retrieved from an authoritative source. For example, an AM may use HTTPS to retrieve information and may check that the domain in the obtained SSL certificate matches that from the XRD/JRD descriptor file. The nonce is included in the file to ensure that it is the actual application that provisions this description.

The dynamic registration module retrieves the $\{client_id, client_secret\}$ pair. These credentials are retrieved only once for each $\{Host, AM\}$ or $\{Requester, AM\}$ pair and are reused among different users of the same application (a user of the application never learns the *client_secret*). For example, when the first user of a particular host application decides to use their chosen Authorisation Manager and this AM is not yet known to the Host then a pair of client credentials is acquired. All further users of this Host would already rely on the previously retrieved client credentials. Importantly, such setup is provided for developer convenience only and there is nothing in the protocol that would prevent the application from obtaining new credentials for each user (although that would not be an efficient mechanism).

UMA/j uses the dynamic registration module to support all aspects of application registration based on the application's configuration provided by the developer. In most cases, this configuration is required to be provided during deployment of the application and not at runtime. Therefore, UMA/j allows developers to provide this information in the XML configuration file. In case of the Pull model, UMA/j provides an endpoint that handles application discovery by the AM. A developer has to include this endpoint in their application without the need of custom implementation (e.g. by including it as a servlet in the `web.xml` deployment descriptor).

Dynamic registration, similarly to discovery, is an optional step and can be omitted by applications that have been pre-registered in any other way (e.g. manually). Additionally, requesters may not be registered at all and can act as anonymous clients, depending on the AM implementation. UMA/j can be configured to allow such anonymous flow. However, in such cases the Requesting Party has to be careful when giving authorisation to an anonymous application (Requester) to access protected resources. This is discussed in more details in Chapter 7.

Dynamic registration, similarly to the discovery module, has been eventually provided as an independent software component. It can be also used in other libraries than UMA/j.

6.2.4 OAuth 2.0

The introduction step in UMA is based on the OAuth 2.0 Authorisation Code Grant [202] and allows Hosts and Requesters to access the functionality provided by an AM. UMA/j supports this step of the protocol with a custom-built OAuth 2.0 library [117; 120; 129] that is integrated with other modules of the framework and can be also used as a standalone library. This library was introduced in Section 1.5.

This OAuth 2.0 library allows an application, which is registered at an AM, to receive either of the following tokens²:

²First versions of the UMA/j framework would not support different token types for requester applications. Instead, the Requester would only obtain a single token that would be used to access protected resources on host applications. This is discussed in more detail in the next chapter of this thesis.

1. **Host Access Token (HAT)** - allows a Host to use AM to register resources which should be protected and to validate **Authorisation Tokens (AT)** received from Requesters;
2. **Requester Access Token (RAT)** - allows a requester application to receive **ATs** that can be used to access protected resources on hosts.

The OAuth 2.0 library can receive the aforementioned access tokens and their corresponding refresh tokens. When a particular **HAT** or **RAT** becomes expired, then a corresponding refresh token is used to obtain a new access token. UMA/j supports this process in a transparent way to the application. The framework provides mechanisms to refresh an access token if the previous one expired based on the AM's response (i.e. when an AM responds with **HTTP 401 Unauthorized**). The framework can also check the expiration of the access token, prior to attempting to use it, and then refresh such token if necessary.

Whether an application receives a HAT or RAT is determined by the *scope* that this application gets authorised for at a particular AM. UMA/j provides default values for UMA scopes (refer to Section 5.4.1 and Section 5.4.4). However, it also allows developers to change the names of these scopes for their convenience and to support arbitrary AM implementations. Moreover, the UMA/j Requester, during its emerging development, has encapsulated the OAuth 2.0 library in one of its own modules (Requester Filter) that is discussed in Section 6.2.6.1.

6.2.5 UMA/j Host

This section demonstrates design of the Host part of the UMA/j framework, called *UMA/j Host*. UMA/j Host is set of modules that allows applications to externalise their access control functionality to Authorisation Managers.

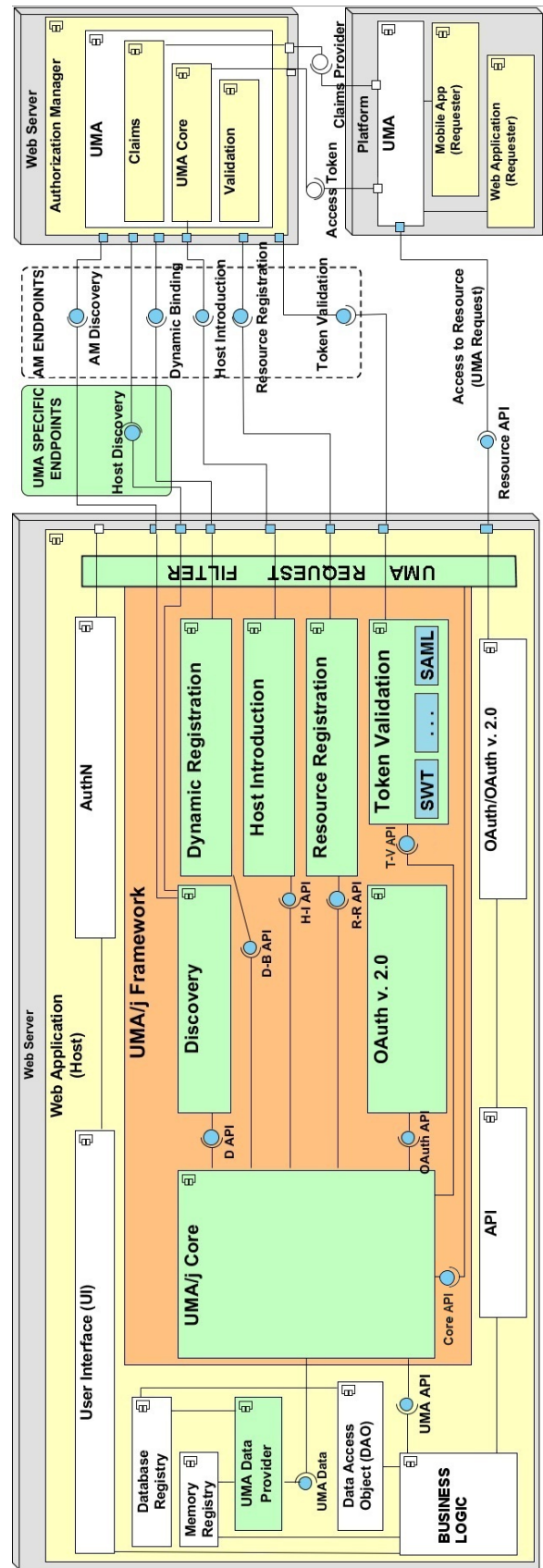


Figure 6.2: Design of the UMA/j framework and its integration with a Web application [239].

UMA/j Host adds host specific functionality on top of the previously discussed modules, i.e. UMA/j Common, Discovery, Dynamic Registration, and OAuth 2.0. This functionality is added in the following new modules:

1. Resource Registration;
2. Token Validation;
3. Request Filter;
4. Core.

All UMA/j modules required by host applications are depicted in Figure 6.2. These modules are discussed in further sections. The dependency graph of UMA/j Host, generated using the Maven Graph Plugin [48], is provided in Appendix A.

6.2.5.1 Resource Registration

The Resource Registration module provides support for flexible authorisation delegation by allowing an application to communicate information about resources to an AM. This information includes resource location and associated metadata (e.g. name, icon). Authorising Users can use this information when defining access control policies at an AM.

This module supports all operations available for the UMA resource registration step. These operations include creating, reading, updating, deleting, and listing resource sets registered at an AM. The module provides a `ResourceRegClient` class that implements an easy-to-use API for the aforementioned operations.

This module allows AMs to communicate the location of the policy URI, which can be used by Hosts for user convenience. In particular, this allows a Host to provide a user with a notion of a "*Sharing Settings*" button. Such button may point to an access control policy associated with a resource and located at an AM.

The Resource Registration module additionally allows a developer to expose a *Resource Registration* endpoint at a host application. Such endpoint acts as an entry point to UMA-related functionality and to the aforementioned `ResourceRegClient`. This endpoint allows for registration and de-registration of resources from AM and is provided as a JAX-RS endpoint [271], which can be included as part of API of an application. Resource registration is then simplified to providing a Web form with a hidden resource ID (e.g. encoded URI of the resource), which should be used during resource registration. When this form is submitted then resource information is used by the discussed endpoint, which takes care of the UMA protocol flow. For example, this endpoint registers a resource or removes one from an AM. In case a resource cannot

be registered, e.g. because the user has not yet introduced a Host to an AM, then the framework can automatically initiate the necessary steps (including discovery, dynamic registration, and host introduction).

A simplified code of a resource registration endpoint is presented in Listing 10. Available actions on resources can be included in the application as static JSON files. UMA/j Host allows for registration of these files using provided configuration options (see `actionId`, `actionUri`, `actionName`, and `actionIconUri` constructor arguments for `UMABasicAction`, as presented in Appendix B). Similarly, deleting resource information from AM can be done using a similar method of the provided JAX-RS endpoint.

```
1  @POST @Path("/resource_reg_entry")
2  public Response createResource(
3      @FormParam("resource_id") String rId,
4      @Context HttpServletRequest req,
5      @Context HttpServletResponse res) throws Exception {
6
7      Principal principal = req.getUserPrincipal();
8      HostUMAINfo info = provider.getUMAINfo(principal, rId);
9      Token at = umaInfo.getAccessToken();
10     if (at == null || StringUtils.isEmpty(at.getToken())) {
11         return performOAuthAuthorization(req, res);
12     }
13     final String token = info.getToken();
14     return performResourceRegistration(rId, info, token, req, res);
15 }
```

Listing 10: Resource registration JAX-RS endpoint for Host applications.

It is important to note that Web applications can register arbitrary resources that can be identified by URIs as required by UMA. The presented framework extends this by allowing arbitrary grouping of resources and protecting such groups by AMs. This is done through UMA-defined notion of *resource sets*. However, a developer is required to define these groups for UMA/j Host. For example, an online gallery service could group photos in albums or allow grouping of individual photos based on user-supplied tags. Similarly, a personal data service could group personal information such as name, phone number, and address to form a user's profile.

A Host may delegate access control to an AM in a very flexible manner, depending on its requirements. As discussed in Chapter 5, a Host can delegate access control for all resources of a particular user, for groups of resources or for individual resources only. Additionally, a Host can delegate access to all its Web accessible resources to a particular AM, too. UMA/j supports

all granularity levels of access control delegation envisioned for the UMA proposal.

6.2.5.2 Request Filter

UMA/j provides a generic mechanism for filtering HTTP requests to Web resources. The Request Filter checks these requests and detects those that are issued to resources protected with UMA. Information whether a resource is UMA-protected is provided by the Core module. A simplified example code for this module is given in Listing 11.

```
1  private HostUMADataProvider dataProvider;
2  private HostUMAVValidationProvider validationProvider;
3  // ...
4  public void doFilter(ServletRequest req,
5                      ServletResponse resp, FilterChain chain) throws Exception {
6
7      if (HttpConstants.Methods.HEAD.equals(req.getMethod())) {
8          UMAResource resource =
9              dataProvider.getUMAResourceFromRequest(req);
10         if (resource != null && resource.isSecured()) {
11             // ... Inform about UMA authorisation
12         }
13         // ...
14     } else {
15         // ... Authorisation Token validation
16         UMAHostResourceRequest resourceRequest =
17             new UMAHostResourceRequest(request);
18         HostUMAAccessDecision decision =
19             validationProvider.validateAccessToken(resourceRequest);
20         if (!decision.isAllowed()) {
21             // ... Require UMA authorisation
22         }
23         // ... Provide access to resource
24         chain.doFilter(req, resp);
25     }
26 }
```

Listing 11: Example simplified code of the Request Filter component.

When the resource is protected but the required **Authorisation Token (AT)**³ is missing then the filter responds with a standard HTTP 401 Unauthorized response and includes the following information about: (1) URL of the AM that protects this resource, (2) the resource identifier as registered at this AM, (3) the Host identifier at this AM. However, if AT is detected, then the filter uses the token validation component, implemented as `HostUMAVValidationProvider`,

³The initially implemented version of UMA/j used the term Authorisation Token instead of Requester Permission Token.

for further processing and AT validation (lines 18-22 in Listing 11).

Importantly, UMA/j does not support permission registration, which was introduced in later stages of the UMA protocol development. Therefore, the described behaviour of filtering access to resources differs from the protocol flow that was discussed in Chapter 5. Instead of registering a permission and returning a permission ticket, UMA/j simply returns information about the protected resource at AM (including resource set identifier, identifier of the host application and the location of the AM).

UMA/j supports a *Security Discovery* mode. Such mode is implemented in the filter and allows requester applications to issue HTTP HEAD requests to hosts integrated with UMA/j. Requesters can investigate whether resources are protected or not, without seeking access to actual resources. Host replies with metadata of a protected resource only, providing the same UMA information as in the case of normal access requests to resources. This mode is configurable by a developer.

The Request Filter module is implemented as a Servlet Filter. This allows the framework to be used in Java Web applications that conform to the Java Servlet specification [101]. This module can be added to an application by including the `UMAHostSecurityFilter` class as a filter in a `web.xml` deployment descriptor of an application.

6.2.5.3 Token Validation

This component supports both local and remote validation of Authorisation Tokens which are first processed by the Request Filter. Importantly, the UMA protocol itself does not restrict the token format to be used. Therefore, this module can work with different token types.

Remote token validation is performed by sending a JSON-formatted access decision request to an AM and receiving a response, which was presented in Listing 9 in Section 5.4.5. This module provides an `UMAVValidationClient` class that implements a `getValidationInfo()` method. This method is used to send an access control decision request to an AM and to retrieve an access control decision response. Such response is then handled by `UMAAccessDecisionEvaluator` that makes the final decision regarding access to a resource. This final decision is made on the Host side.

On the other hand, local validation is performed by plugins responsible for different token types. By default, the framework supports validation of SWT tokens [115] by validating signatures, using the supplied public key of the AM, checking the token's issuer (i.e. if it is the AM that was registered by a user) and matching its properties against requested access type. UMA/j also supports the inclusion of custom plugins for other token types.

6.2.5.4 Core

This Host module provides the following functions:

1. Allows a host application to use UMA functionality;
2. Provides services for all UMA/j components;
3. Exposes an abstraction of persistent storage of UMA-related data to these components, based on the code provided by the UMA/j Common module.

This module combines all the Host modules together. It provides the configuration options, i.e. allows the applications to attach the UMA/j Host framework and configure its different aspects. An example of UMA/j Host configuration, included in the Spring XML metadata configuration file, is given in Appendix B.

The Core module hides the complexity of using separate UMA/j components by exposing a high-level *UMA API*. Firstly, this API allows an application to delegate access control to an AM. Secondly, it allows to an application to register resources that are meant to be protected. The framework then takes care of the underlying steps (e.g. *Discovery*, *Dynamic Registration*, and *Introduction*).

This module is used by other UMA/j components. For example, these components use Core to obtain discovered information about endpoints or get the correct **HAT** that would be used for resource registration and token validation.

The Core module provides an interface, called `HostUMADataProvider`, for persistent storage of all UMA-related data which includes endpoint descriptions, credentials, and access and refresh tokens for different AMs (possibly used by different users of the same Web application). This information is then used by an application and other modules of the framework. UMA/j depends, however, on a developer to provide a concrete implementation of this interface (e.g. a developer can decide to store UMA-related data in a file or using a SQL-based database). The UMA/j Core module provides implementation of a `MemoryHostUMADataProvider` for testing purposes and this implementation stores data in a `HashMap`.

6.2.6 Requester

This section discusses design of the Requester part of the UMA/j framework, called *UMA/j Requester*. This set of modules allows applications to access UMA-protected resources at Hosts.

UMA/j Requester extends the common UMA/j functionality by adding the following new modules:

1. Requester Filter;

2. Token Retrieval;
3. Core.

The dependency graph of UMA/j Requester, generated using the Maven Graph Plugin [48], is provided in Appendix A.

Currently, the framework is targeted at Web applications only, which can use it through the provided *UMA API*. Future plans include adaptation of the framework for applications running on mobile devices. Similarly to UMA/j Host, applications can also use low-level APIs, in case more fine-grained control of the protocol flow is required. Importantly, UMA/j Requester is simpler in comparison to UMA/j Host. This section focuses on a late version of the framework with already implemented support for OAuth 2.0 and RAT tokens. Initial versions of UMA/j Requester only needed to obtain Authorisation Tokens from an AM (see SMARTAM V1 discussed in Section 7.2) and could use these tokens to access protected resources.

6.2.6.1 Requester Filter

Requester Filter is the main component used in the framework for Requester applications. It is applied as a filter to the actual business logic implementation, e.g. servlet or controller. The role of this filter is to intercept all User-Agent-initiated calls to an application to access resources, which may be UMA-protected, before these calls reach the actual business logic of the application (e.g. logic of the implemented HTTP client in the application). The filter uses the `UMAMappingProvider` to check if a requested resource has been previously accessed by this application and if this resource was UMA-protected.

If the resource was previously accessed, then Authorisation Token information is obtained from the provided implementation of the `RequesterUMADataProvider` interface. This module checks if the AT is valid and it can refresh this token as necessary. In case a refresh token was not issued by the AM in the first place, the framework directs a user to an AM according to the OAuth 2.0 protocol, as presented in Section 6.2.6.2. Earlier versions of the UMA/j Requester framework did not use the OAuth 2.0 Web Server Flow (Authorisation Code Grant) but were based on the OAuth 2.0 Username and Password flow [202]. The latter approach in UMA/j Requester was used during integration of applications with SMARTAM V1 but this approach is not discussed further in this thesis.

However, if a resource has not yet been accessed, then the filter executes a normal UMA flow. In particular, it can send the HTTP `HEAD` request to the resource. The purpose of this request is to check whether a resource is protected with the UMA protocol (recall the Security Discovery mode of the UMA/j Host framework, which was discussed in Section 6.2.5.2). If the

resource is protected, then the host's response should contain all the necessary information to proceed with the UMA flow, i.e. URL of the AM protecting the resource, the resource ID, and the Host ID (this differs from the original UMA proposal - recall Section 5.4.4). Requester Filter does not pay attention to the actual AM that protects the resource because this function is covered by the Token Retrieval module (see Section 6.2.6.2).

Importantly, the implemented filter does not access the resource itself but directs the original User-Agent-initiated request, along with the attached AT, to the actual controller/servlet. Such controller/servlet then uses its own HTTP client to access a resource. Therefore, UMA/j allows applications to use their own HTTP client libraries and still benefit from the provided UMA functionality.

Requester Filter is developed as an ordinary Servlet filter in the `RequesterResourceFilter` class. Such implementation allows this component to be applied to any Java Web application that conforms to the Java Servlet specification [101].

6.2.6.2 Token Retrieval

The Token Retrieval module is provided to support application developers with obtaining Authorisation Tokens. This module consists of three parts: `RequesterTokenRetrievalServlet`, `RequesterCallbackServlet`, and `ClaimsProcessor`. The `RequesterTokenRetrievalServlet` is used to initiate the UMA protocol flow. On the other hand, the `RequesterCallbackServlet` is used to obtain authorisation for an AM in form of a short-lived authorisation code. Both `RequesterTokenRetrievalServlet` and `RequesterCallbackServlet` encapsulate OAuth 2.0 (recall Section 6.2.4). The authorisation code is eventually exchanged for RAT using the AM's Token Endpoint. UMA/j also provides a `ClaimsProcessor` implementation, which supports processing of *claims-requested* and *claims* documents according to the Claims 2.0 specification (recall Section 5.4.4). The `ClaimsProcessor` was used in early versions of the framework and is discussed only briefly in this section.

The Token Retrieval module is used in UMA/j in two cases. Firstly, if the resource being accessed is not yet known and the Requester Filter detects that this resource is UMA protected, then the `RequesterTokenRetrievalServlet` is used to obtain authorisation from the correct AM to access that resource. In particular, it may perform the *Discovery*, *Dynamic Registration*, and *Introduction* parts of the UMA protocol flow (these functions were discussed in Sections 6.2.2, 6.2.3, and 6.2.4 respectively). This module can then obtain the Authorisation Token. Secondly, this module is used in case the RAT (or the AT), which have been previously obtained from the AM, are expired or invalid and there were no corresponding refresh tokens issued. For example, if a RAT is expired then this module redirects a Requesting Party to an AM to obtain

a new code that is later exchanged for a RAT.

In order to obtain a RAT using the OAuth 2.0 protocol, this part of the framework provides the `RequesterCallbackServlet` class. When a Requesting Party is redirected back from an AM to a Requester, after authorising this application for an AM, then such redirect is accompanied with a short-lived authorisation code. This code is exchanged for a RAT that is used at the AM to obtain the necessary AT.

The **Authorisation Token** is used for accessing protected resources at Hosts. A Requester can obtain an AT from an AM. How the Requester communicates with AM in order to obtain authorisation has changed over time. In early versions of the framework, the Requester provided the URL of the resource that it wanted to access and the type of access (e.g. HTTP GET method). UMA/j was further adapted to support providing resource and host identifiers to an AM. Moreover, the early versions of UMA/j Requester did not differentiate between tokens used for communication with AM (RATs) and tokens used for accessing protected resources (ATs). A Requester had to authenticate a Requesting Party to an AM (similarly to OAuth 2.0 Username and Password flow). Support for both token types has been eventually introduced and support for RAT has been added to UMA/j.

Because of the changing UMA protocol and the UMA/j implementation, as discussed earlier, these evolving parts of the framework are not shown in detail. Instead, a more detailed integration between a Requester and an AM is presented on example of a second framework, which is demonstrated in Section 6.3 because this framework implements a later, improved revision of the UMA proposal.

As discussed in Chapter 5, user policies at an AM may define claims, which must be submitted by Requesting Parties before an **AT** can be provisioned. The implemented Token Retrieval module contains a `ClaimsProcessor` component that implements an amended version of the Claims 2.0 specification. This component allows parsing *claims-requested* documents received from an AM and presenting them to a Requesting Party during the authorisation phase.

Developers are required to provide a UI view for presenting such requested claims, as well as for gathering user's consent to those claims. For example, the developer must implement a view to ask the user to confirm that they are over 18 years old and then record the confirmation. Users must select claims in order to confirm them. These claims are later processed by `ClaimsProcessor` and are sent to the AM. Only then the AT can be provisioned by AM or, if further claims should be submitted, a new *claims-requested* document is received. Importantly, UMA/j supports only self-asserted claims, which is one of the limitations of this framework.

6.2.6.3 Core

This module of the UMA/j Requester framework provides a similar function to its equivalent in UMA/j Host. In particular, its aim is to:

1. Provide services for all UMA/j components;
2. Expose an abstraction of persistent storage of UMA-related data to these components, based on the code provided by the UMA/j Common module.

The Core module provides functionality to other Requester modules. Similarly to UMA/j Host, this module provides the configuration options, i.e. allows an applications to attach the UMA/j Requester framework and to configure its different aspects. Such configuration is similar to the UMA/j Host configuration shown in Appendix B.

This module provides services for other UMA/j components that can use Core to get the required data. Core exposes a `RequesterUMADDataProvider` interface, which supports storing and retrieving information about authorisations to resources. Developers have to implement this interface in their applications but UMA/j provides a basic in-memory implementation, too.

The Core module also includes `UMASstateProvider` and `UMAMappingProvider` interfaces. The first interface declares methods used for storing state information that defines a resource accessed by a Requesting Party. This information must be remembered before a Requesting Party is redirected to an AM and must be restored when a Requesting Party is redirected back to a Requester. The latter interface allows the framework to store $\{resourceUri, hostId, resourceId\}$ tuples for UMA-protected resources. This information allows the framework to establish if a resource is protected before issuing any requests to the Host. UMA/j provides in-memory implementations of both interfaces.

6.2.7 Limitations

Integration of host and requester applications with the UMA/j framework still required a significant level of knowledge of the underlying protocol, which was considered a major issue with this software. Furthermore, the UMA protocol was continuously improved and some of the protocol flows changed over time. This affected host and requester behaviour, e.g. permission registration and permission tickets were added to the protocol. Moreover, obtaining that way tokens were used in the software changed over time as well (recall Section 6.2.6.2).

Introducing changes to the UMA/j framework required a significant effort and each change triggered further adjustments in unit and integration tests. Using Java for further development of Host or Requester applications, along with the necessary tests, was not consider pertinent at

that stage of the rapid development of the UMA protocol. Therefore, features such as support for third-party asserted claims or permission registration were not added to the framework despite the fact that this part of the UMA protocol was considered mandatory in further research. Moreover, retrieving authorisation tokens was not finalised and integration between a Requester and AM was tested only in a limited form.

UMA/j Host was tested primarily with bearer tokens which were opaque to host applications. Therefore, UMA/j was mostly concerned with remote token validation. Such validation may not be sufficient in terms of performance for applications where access to resources is frequent. Although UMA/j provides initial implementation of local token validation, such validation was not tested further during integration of UMA/j with example Java applications (these applications are discussed in Appendix C).

Based on the experience gained with UMA/j, a new framework was implemented in Python. The goal of the new framework was to address the aforementioned challenges as well as implement new features that were introduced in the UMA proposal.

6.3 Puma

The Puma⁴ framework has been developed to further extend the possibility of integrating Web applications with UMA Authorisation Managers in different environments than Java. As presented in Table 6.1, this framework differs in several ways from UMA/j.

Firstly, developers can use Puma to build UMA-enabled Host and Requester applications. Puma does not support development of Authorisation Managers, while such functionality is present in the UMA/j framework. Puma was developed according to a newer revision of the UMA protocol and it was tested against SMARTAM V2. In particular, Puma implemented support for permission registration and permission tickets. It also supported self-asserted and third-party asserted claims through redirects. Puma adopted the new terminology of the UMA proposal and used Requester Permission Tokens and not Authorisation Tokens. Furthermore, it is targeted at developers familiar with the Python programming language and those who want to build Web applications using the Google App Engine (GAE) platform [21] (the framework eases integration of GAE applications with UMA but can be effectively used with other Python frameworks, such as Django [8], as well).

In terms of programming paradigms, the Python UMA library approaches a functional design [295]. Such functional programming focuses on the application of functions, in contrast to imperative programming that puts emphasis on changes in state [208].

⁴The name *Puma* originally came from combining the words Python and UMA, also indicating the ability of very rapid development of UMA-enabled Web applications 'at a speed of a Puma animal'.

Subsequent sections of this chapter discuss the design of Puma and its integration with example UMA-compliant Host and Requester applications. These applications have been built taking into account the scenario, as presented in Section 1.1.2. Puma has been designed and implemented in a way, which allows its use by various Web applications. The discussion in this section follow steps of the UMA protocol, i.e. each section represents a part of the UMA flow and maps the designed modules and functions of the framework to this flow. Parts of this section were first published as blog posts in [123] and [124], as a joint effort with Jacek Szpot.

Section 6.3.1 discusses the framework's structure and its modular design. Section 6.3.2 introduces the common functionality provided by Puma. Host and Requester parts of the framework are discussed in Section 6.3.3 and Section 6.3.4 respectively. Limitations of the Puma framework are presented in Section 6.3.5. Puma evaluation and implementation is given in Appendix D.

The host related functionality of Puma is presented using an example of a simple Personal Data Store (PDS) type application implemented using Google's Python `webapp` framework. The PDS application resembles the previously discussed S3P application [69]. The Puma Requester part is presented using a simple UMA-enabled client application that can retrieve Web-accessible resources according to the UMA protocol (this application has been named CareerMonster and provides a similar function to an online job application portal for graduate students allowing them to apply for advertised job positions).

6.3.1 Framework Structure

Objects and functions in Puma are categorised depending on their role in the UMA protocol. These categories are shared between protocol steps that are common, as well as between steps that are specific to Host and Requester related functionality. This categorisation is visualised in Figure 6.3. A description of each category is given further in this section.

6.3.1.1 Puma.UMA.*

Functions related to core UMA functionality, irrespective of the role of the application (i.e. whether it acts as a Host or Requester), reside in `Puma.UMA`. This includes functions such as `discover_am_endpoints(...)`, `register_resource(...)`, `check_token_status(...)` and other methods common for UMA clients.

6.3.1.2 Puma.OAuth.*

This module contains OAuth 2.0 parts of UMA which are specific to the framework. Among others, the following functions are available in this module: `get_authz_uri(...)`, `trade_code_for_hat_and_store(...)` and a `refresh_token(...)` function.

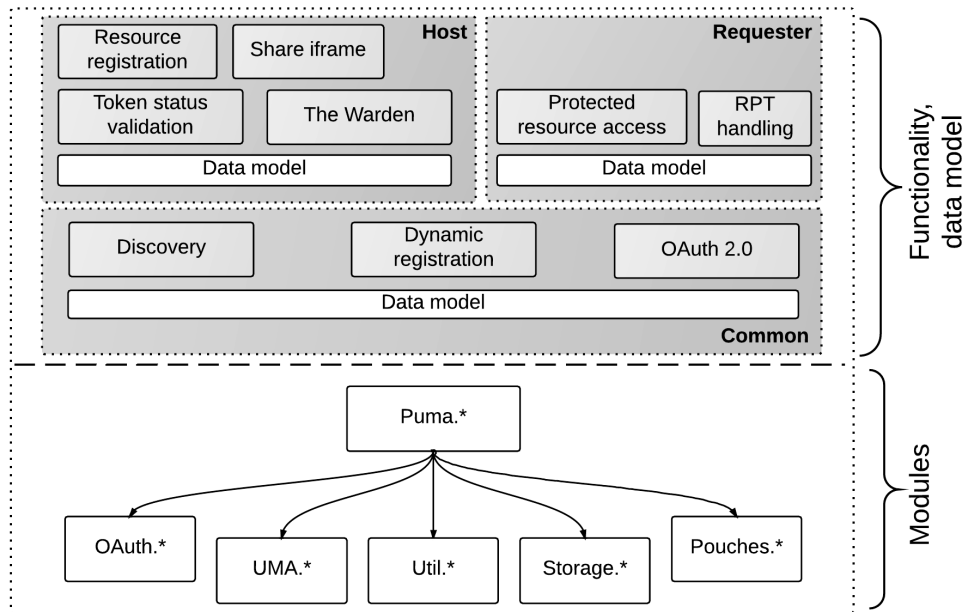


Figure 6.3: Structure of the Puma framework [123; 124].

6.3.1.3 Puma.Util.*

Utility and helper functions used both internally and externally by the library are available in the `Puma.Util` module. An example function is `get_owner_of_resource(...)`.

6.3.1.4 Puma.Storage.*

This module provides an abstraction layer for allowing Web applications that integrate Puma to use different persistent storage technologies. This module provides, among others, the following functions: `get_uma_user_for_user(...)`, `create_new_uma_user_with_am(...)` and a `get_am_by_hostname(...)` function.

6.3.1.5 Puma.Pouches.*

This module provides a collection of objects that represent data and messages passed between client applications and AM. These objects have been provided with such methods as `to_json()`, which allows for consistent JSON encoding of messages in Puma, e.g. `RegistrationData` and `ResourceSetDescription`, which represent client application information as well as resource set that is registered by a Host respectively.

6.3.2 Common

The common part of Puma contains the functionality which is required by all UMA client applications, irrespective of whether these applications act as Hosts or Requesters. In particular, it contains functionality related to common parts of the UMA protocol flow, i.e.:

1. Discovery (Section 6.3.2.1);
2. Dynamic registration (Section 6.3.2.2);
3. OAuth 2.0-based authorisation (Section 6.3.2.3).

6.3.2.1 Discovery

Applications that want to interact with AM must first learn about its API endpoints, as discussed in Chapter 5. Host applications must be able to discover the *Protection API* and requester applications must discover the *Authorisation API*.

Puma provides a `discover_am_endpoints()` function that performs discovery and returns a configuration for a given AM. This function returns an `AuthorizationManager` entity that is created in the persistent datastore. This entity serves as a binding point and copy of the JRD-based `uma-configuration` file for that AM.

The locally created data is kept in synchronisation with the contents of the original JRD file. This process, however, is "manual". The JRD document is synchronised when the same AM is chosen by another user (for Host applications) or when a resource is accessed and simultaneously protected by this AM (for Requester applications).

Currently, neither UMA nor the presented AM implementations contain an extension that would allow the JRD configuration file to contain a validity (or refresh) period. After such period, the client application would need to check if the contents of the file have changed. At this moment, this is only achieved using the existing HTTP-based mechanisms (e.g. `Cache-Control` or `Expires` headers of the HTTP response with the `uma-configuration` file [186]).

6.3.2.2 Dynamic Registration

Dynamic registration has been implemented according to one of the early drafts of the OpenID Connect Dynamic Client Registration protocol⁵ [264] and not using the latest OAuth 2.0 Dynamic Client Registration [274]. The client application issues a *registration request* and receives OAuth 2.0 client credentials. Application information can be later updated, if necessary.

⁵At the time of writing, Puma implementation was not fully compatible with this standard.

Dynamic registration is supported by the `Puma.OAuth` module and its `oauth_registration()` function. This function accepts a `RegistrationData` object. Such object needs to be created by a developer and must contain information about the client that will be registered at an AM. Upon successful registration, the client application is provisioned with client credentials, which can be later stored in a datastore. Storing data can be done using the provided `Puma.Pouches` module which binds the AM configuration to a particular user. Registration is done once per application and not per user, and client credentials and the configuration can be later shared with other users that choose to interact with the same AM, either through a Host or Requester application.

6.3.2.3 OAuth 2.0-based Authorisation

Hosts and Requesters require a valid token to interact with the *Protection API* and the *Authorisation API* respectively. A Host needs to engage an *Authorising User* in order to obtain a token for the Protection API. Such applications then act on behalf of an Authorising User when registering resources or validating tokens at an AM. Requesters need to engage a Requesting Party to obtain a token for the Authorisation API. These applications act on behalf of a Requesting Party when providing necessary claims to an AM.

Obtaining authorisation in form of an access token, either **Host Access Token** or **Requester Access Token**, is a standard OAuth 2.0 flow. A client application redirects an end-user to a *User Authorisation* endpoint at an AM. The location of this endpoint is obtained from the `uma-configuration` JRD file.

Puma handles the redirect of a user to a valid URL of a *User Authorisation* endpoint at an AM. It prepares this URL, with a set of client-specific parameters, using the `authz_uri_for_user()` function. Puma uses the following scopes for Hosts and Requesters respectively: `uma_host` and `uma_requester`. Both scope names are configurable and can be changed to the recently adopted scopes in UMA (refer to Section 5.4.1 and Section 5.4.4 for more details).

When a user authorises a client application for an AM, this user is redirected back to this application with a code. The callback handler at the client application exchanges this code for an access token. Puma provides the following functions that allow to exchange a code for an access token and that can be used by Hosts and Requesters respectively: `trade_code_for_hat_and_store()` or `trade_code_for_rat_and_store()`. The obtained HAT or RAT is later used by a client application to interact with an AM.

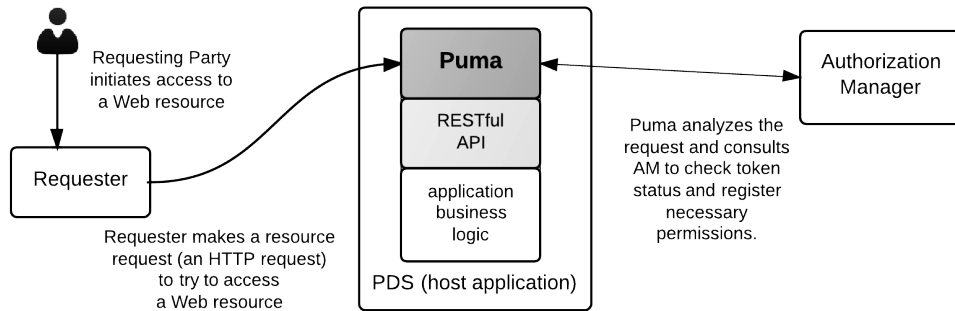


Figure 6.4: High-level architecture of Puma integration with a host application [123].

6.3.3 Host

This section demonstrates functionality of the Puma framework that is available for host applications. This part of the framework is referred to as Puma Host in this section. The given discussion follows presentation of an implemented simple Personal Data Store (PDS) type application. Even though the PDS is used as an example of a Host, the same principles apply to any host application that integrates with Puma. The PDS application is discussed further in Section D.0.4.1 of Appendix D.

The host functionality provided by the Puma framework, along with its typical elements and mechanisms, is discussed in the following sections:

1. Data model (Section 6.3.3.1);
2. Resource registration (Section 6.3.3.2);
 - (a) Resource registration handler (Section 6.3.3.2.1);
 - (b) Share button iframe (Section 6.3.3.2.2);
3. Token status validation (Section 6.3.3.3);
 - (a) The Warden (Section 6.3.3.3.1).

The provided descriptions follow the flow of the UMA protocol, which is used by host applications. These descriptions assume that discovery, dynamic registration and Host introduction have been completed.

6.3.3.1 Data Model

Host applications that integrate with Puma must have some notion of *users* or *accounts*. Users in such applications must have locally unique identifiers. Puma Host references these identifiers in its own data model. The relationship between Puma's own entities and entities of the application

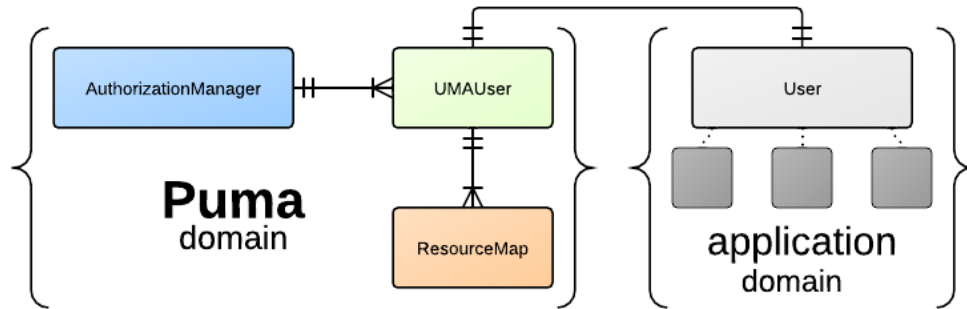


Figure 6.5: Simplified ERD diagram of using Puma framework in a host application [123].

is visualised in Figure 6.5. This figure shows the application-specific notion of *users/accounts* as the **User** object. This object is the sole point of integration between the application and the framework. Puma objects are defined below:

AuthorizationManager

These entities store information about discovered authorisation managers. Each entity holds data that is related to the discovery and dynamic registration phase of a particular AM. The former is a list of AM endpoints, i.e. Protection API, while the latter is a set of OAuth 2.0 credentials.

ResourceMap

A single **ResourceMap** entity represents and holds information about a resource on a host, which has been registered for protection. This information includes, among others: resource ID, URI, name, icon URL, and an optional short description for UX purposes.

UMAUser

The previous two entities are related together through **UMAUser**. This entity allows Puma to get information about an AM that protects resources of a user, which user is an owner of a **ResourceMap**, etc. This entity has a monogamous relation (*has*) with an **AuthorizationManager** and an outside **User** entity. It also has a polygamous (*has-many*) relationship with **ResourceMap**.

6.3.3.2 Resource registration

Puma provides a `Pouches.Puma.ResourceSetDescription` object which is used by developers during resource registration. This object has to be created with information about a resource and is then converted into a JSON description (see Listing 12). The `pre_generated_id` is a unique identifier chosen by a Host and this identifier is used by an application to reference a resource at an AM. Resource registration is handled by the `register_resource(...)` function.

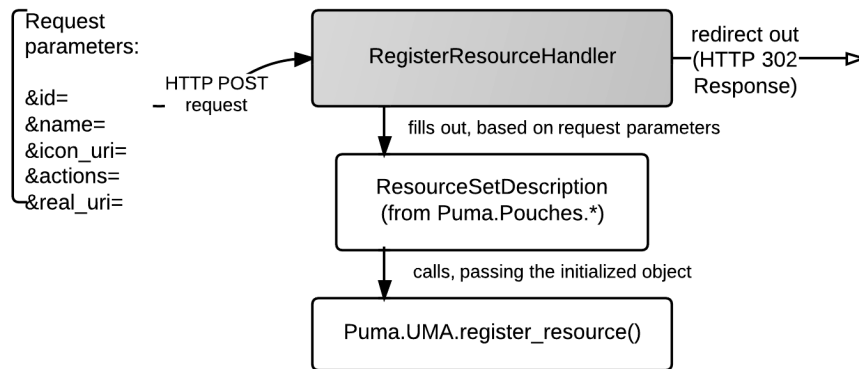


Figure 6.6: Flow using Resource Registration Handler [123].

Once a resource is registered at an AM, the final step involves creating a `ResourceMap` object in the local datastore (Listing 12). This object stores information about the registered resource. In particular, it contains the location of the access control policy for this resource. The presence of a `ResourceMap` is how Puma determines whether a resource requested via the RESTful API is UMA-protected.

```

1  res_description = Puma.Pouches.ResourceSetDescription()
2  res_description = # ...fill in details of resource description ...
3  rx = Puma.UMA.register_resource(user_key, res_description)
4  if rx.status == 201: # resource registered successfully
5      json_data = json.loads(response.read())
6  else: # registration failed
7      ...
8  etag = response.getheader('ETag')
9  policy_uri = json_data['policy_uri']
10 map = Puma.Util.create_resource_map(real_uri, pre_generated_id,
11                                     etag, name, icon_uri, policy_uri, user_key)

```

Listing 12: Registering a resource set at AM and creating a `ResourceMap` on a Host application.

6.3.3.2.1 Resource Registration Handler. In order to simplify the process of resource registration, Puma supports the use of a resource registration handler, which is similar to the UMA/j Resource Registration endpoint discussed in Section 6.2.5.1. The PDS can delegate the resource registration process to a specially constructed handler that resides at the `/uma/register-resource` path (and under the context of where the application is deployed). A flow chart describing the resource registration process using the proposed handler is shown in Figure 6.6.

This handler expects a set of values through an HTTP POST request. Therefore, resource registration can be simplified to providing a standard Web form with a hidden resource ID (e.g.

encoded URI of a resource), which should be used during resource registration at an AM. When this form is submitted by a user then resource information is used by the handler to create a `ResourceSetDescription` object. This object is then used as an argument for the `register_resource(...)` function that registers the resource at the user's AM. Upon receiving a success response, a local resource description is created in the form of a `ResourceMap` object that is stored in a persistent storage. The handler then redirects a user to a specified callback address at an application and this allows for the resource registration process to be transparent for a user.

6.3.3.2.2 Share button iframe. Puma extends the core UMA proposal and allows a user to view an access control policies for a protected resource when the user is at a host application. This is achieved through the use of an `iframe` element. Puma provides a handler, called `UMAMiniPanelHandler`, that is meant to be used inside an `iframe`. This handler provides a clean separation of the framework and the business logic of the application.

This handler can display either of the following buttons:

Share - displayed when a resource is not yet registered at an AM;

Sharing Settings - displayed when a resource is already registered at an AM.

When a resource is protected by an AM, the handler also displays the following button:

Stop Sharing - displayed along with the *"Sharing Settings"* button to allow de-registration of a resource from an AM.

The `UMAMiniPanelHandler` handler can be made available at `/uma/uma_mini_panel` path of the application and its input parameters include the required URI and the optional name of a resource. An example simplified source of an `iframe`, used by the PDS application, that uses the jQuery library [36] to interact with the handler is given in Listing 13.

Upon receiving an HTTP request from a Web browser, the handler checks if the resource identified by the URI is protected with UMA. The handler then retrieves the policy URI for this resource from the persistent store (recall the `ResourceMap` object) and creates a *"Sharing Settings"* button that is displayed on a Web page. This button points to the access control policy and it can be used by a user to access this policy directly from the Host. The handler also generates an additional *"Stop Sharing"* button.

On the other hand, if the resource is not protected, then this handler creates a *"Share"* button. When this button is clicked, a hidden Web form containing the resource information is submitted using HTTP POST to the `/uma/register-resource` URL. The resource is registered


```
<h3>Phone number</h3>
<!-- Puma start. -->
<div style="display:inline;" id="uma_panel_phonenumber">
  <script>
    $.get('/uma/uma_mini_panel?
      uri=/api/users/{userid}/phone&name=Phone%20number',
      function(data) { $('#uma_panel_phonenumber').html(data); });
  </script>
</div>
<!-- Puma end. -->
<p>Your phone number is <b>{{phonenumber.number}}</b></p>.
```

Listing 13: Dynamically generating a widget for integration with Authorisation Manager.

at the user's preferred AM and only then the *"Sharing Settings"* and *"Stop Sharing"* buttons are generated.

To simplify the user experience for sharing resources directly from host applications, a single button (*"Sharing Settings"*) has been eventually used. This button allows a user to register a resource (if necessary) and view an access control policy with a single click.

6.3.3.3 Token status validation

In UMA, any requests made by a Requester come with an attached RPT that is included in a WWW-Authenticate header (recall Section 5.4.5). The token itself can be an opaque value to the Host. In such case, a Host has to refer to an AM in order to validate such token.

Puma provides the `Puma.UMA.check_token_status()` function that performs token validation. This function is used when the framework checks if the requested resource is indeed protected by any AM (see Section 6.3.3.3.1 for more details). Based on the information included in the AM's token status response, the host application can decide whether access to a particular resource should be granted. For example, the Host translates the `/users/123/phone` URL to a resource ID that is managed internally. The framework then compares this ID with the one that is returned by an AM in the token status response. Furthermore, the Host translates the HTTP GET request to the `https://puma-pds.com/scopes/read.json` action and also compares if such action is present in the token status response.

Importantly, the application must know how to translate actions to scopes (and vice-versa) for its resources. Such translation is configured by the developer of the application. In Puma, the entire process of validating RPT tokens is encapsulated in the `Warden` component, which is discussed in the next section.

6.3.3.3.1 The Warden. The Warden component implemented in Puma provides token validation function. This component is a piece of WSGI⁶ middleware that can be used with WSGI-compliant Python Web applications. It acts as a filter to incoming requests (recall Figure 6.4).

Typically, an application (such as the discussed PDS) is composed of the following two parts: (1) the User Interface (UI), and (2) the Application Programming Interface. The PDS implements these two parts in two separate files:

main.py

This file consists of the native application code, i.e. business logic of the PDS application related to storing personal data, used by the User Interface.

restapi.py

This file contains the implementation of the Web API that allows read-write access to data stored by the application. The API follows RESTful principles and maps HTTP verbs (e.g. GET, POST, PUT, DELETE) to actions on resources.

Any HTTP request that is directed to `/api` path is processed by the code in `restapi.py` and all other requests are processed by the code in `main.py`.

The Warden component intercepts requests to an API and applies UMA-based authorisation. In particular, it can check if a resource is protected and whether access to this resources should be granted. If a `ResourceMap` object for the resource's URI exists in the datastore then the resource is considered protected with UMA. A simplified example code of the Warden component is given in Listing 14. The flow of this component is illustrated in Figure 6.7.

Case one (line 4): if a `ResourceMap` object is found, the Warden acts accordingly to the UMA protocol - check for an access token, ask the AM about the status of the token, optionally register permissions at the AM, respond to the Requester with an UMA response (with optional ticket) and continue according to the protocol;

Case two (line 16): if a `ResourceMap` object is not found, then the resource is not considered UMA protected. The request is passed to the API handler, which prepares a response based on its own business logic and responds back to the Requester.

Puma implementation differs slightly from the recent revisions of the UMA protocol. The permission ticket is not returned in the body of the response to the Requester. Instead, Puma returns such ticket in the `WWW-Authenticate` header, which must be extracted by the requester application in order to seek access to a protected resource (refer to line 12 in Listing 14). Future plans include adjusting Puma to the most recent revision of the UMA protocol.

⁶WSGI is the Web Server Gateway Interface. It is a specification for web servers and application servers to communicate with web applications and it is also a Python standard, described in PEP 333 [89; 62].

```

1  path = environ['PATH_INFO']
2  map = ResourceMap.gql("WHERE real_uri = :1", path.strip('/')).get()
3  if map:
4      ... # resource is protected with UMA
5      token_status_re = UMA.check_token_status(am, hat, request_body)
6      body = token_status_re.read()
7      verdict = self.check_if_enough_permissions(
8          resource_id, environ['REQUEST_METHOD'], body)
9      if not verdict:
10         ticket = UMA.request_permission(rat, resource_id, owner,
11             self.permissions_map[environ['REQUEST_METHOD'].lower()])
12         build_headers(host_id, resource_id, am_url, ticket)
13         start_response('403', headers)
14     else:
15         ... # access to a resource is granted
16 else:
17     ... # resource not UMA-protected
18 return self.app(environ, start_response)

```

Listing 14: Puma Warden logic that checks if a resource is UMA-protected (simplified code) [123].

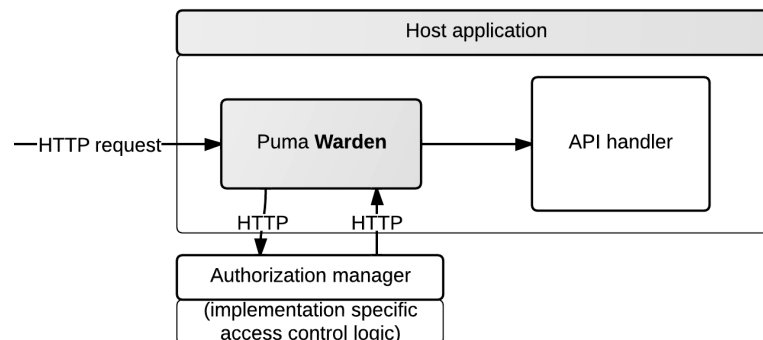


Figure 6.7: Message flow through the Puma Warden [123].

6.3.4 Requester

This section demonstrates functionality of the Puma framework that is available for requester applications. This part of the framework is referred to as Puma Requester in this section. The given discussion follows presentation of an implemented simple client application.

The application, named CareerMonster, is implemented as a demo of an online job application system. CareerMonster allows students to create their personal accounts where they can apply for advertised job positions. The application process requires submitting personal information, such as email address, full name, as well as the verified "Transcript of Records" document (recall the scenario presented in Section 1.1.2).

Additionally, the CareerMonster application allows prospective employers to create accounts

and add new job positions. Importantly, employers can later view applications submitted by students and can request additional data from students. For example, employers can request access to the phone number information stored at the PDS.

CareerMonster can access UMA-protected resources using the Puma framework. In particular, it allows students to import their protected data from the PDS application. Additionally, it can request access to UMA-protected resources, i.e. request for access is submitted by a Requesting Party and later evaluated by an Authorising User. Such functionality is currently not a part of the UMA protocol but was added as an extra feature to this framework. This feature is presented in Section 7.4.6. This application is further discussed in Section D.0.4.2 of Appendix D.

Puma Requester is demonstrated in the following sections:

1. Data model (Section 6.3.4.1);
2. Protected resource access (Section 6.3.4.2);
3. Obtaining Requester Access Token (Section 6.3.4.3);
4. Obtaining Requester Permission Token (Section 6.3.4.4).

Given descriptions follow the flow of the UMA protocol that is used by requesters. Even though the CareerMonster is used as an example of a Requester in this section, the same principles apply to any requester application that integrates with Puma.

6.3.4.1 Data Model

Requester applications must have some notion of *users/accounts* in order to integrate with Puma. These users must be identifiable by a locally unique ID. The relationship between Puma's own entities and that of the Requester is visualised in Figure 6.8. This figure demonstrates the application-specific notion of *users/accounts* as the **User** object. This object is the sole point of integration between the application and the framework. Other objects are discussed below:

AuthorizationManager

These entities store information about known (i.e. discovered) authorisation managers, which protect resources at hosts (recall Section 6.3.1.4). For requester applications, only endpoints of the AM's Authorisation API have to be known.

RequesterAccessToken

A **RequesterAccessToken** entity represents a single RAT that is required by a Requester to interact with an AM on behalf of the Requesting Party. A **Requester Access Token**

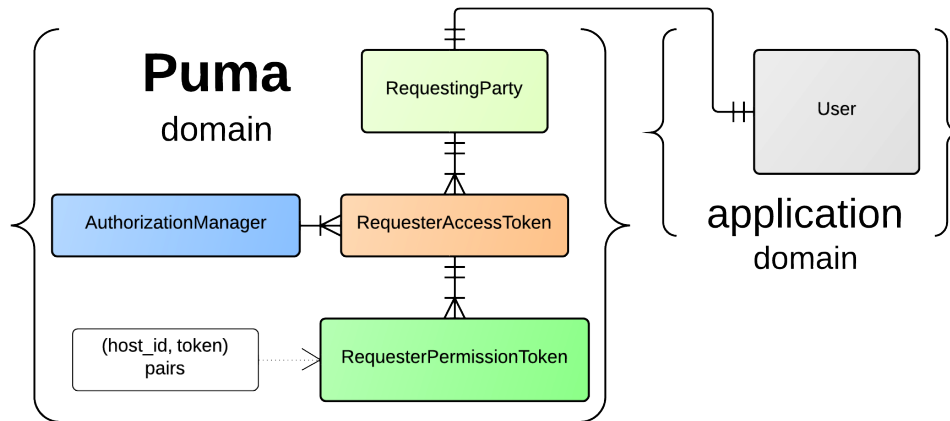


Figure 6.8: Simplified ERD diagram of using Puma framework in a requester application [124].

is a regular OAuth token. There is only one token per $\{Requesting\ Party, Authorisation\ Manager\}$ pair. This token allows a Requester to access the AM's *Authorisation API*. A `RequesterAccessToken` entity is linked to an `AuthorizationManager` entity that represents the AM for which this token is valid. This entity has as a polygamous (*has-many*) relationship with a `RequesterPermissionToken`.

RequesterPermissionToken

A `RequesterPermissionToken` entity represents a single RPT obtained by a requester application for a particular host. A Requester requires at least one token per Host (but can support multiple tokens) and obtains these tokens from authorisation managers. A Requester has to be in the possession of a RAT to obtain RPTs. These RPTs can be associated with permissions for specific resources on Host applications.

RequestingParty

This entity represents a user in Puma. It allows the framework to obtain information about RATs that can be used to interact with authorisation managers. A `RequestingParty` entity also allows Puma to obtain information on RPTs that could be used to access protected resources. This entity has a monogamous relation with the application's `User` entity; as well as a polygamous (*has-many*) relationship with `RequesterAccessToken`.

Puma provides a set of API calls to support management of RAT and RPT tokens. Therefore, developers are not required to deal with these tokens directly and can focus on the functionality of their requester applications.

6.3.4.2 Protected Resource Access

Before a Requester, on behalf of a Requesting Party, can access a protected resource, it can check if such resource is indeed UMA protected. Puma supports Security Discovery that provides such functionality (recall the UMA/j framework demonstrated in Section 6.2.5.2). The framework can make an initial HTTP request (in the form of HTTP **HEAD** request) to a resource. Puma expects such an initial request to fail (unless a Requester already has the RPT in place). However, headers (especially **WWW-Authenticate**) of the response should contain the necessary information for the UMA protocol to be further executed. In particular, Puma determines if a resource is UMA protected based on the following:

1. HTTP response status is HTTP 401 **Unauthorized**;
2. HTTP response contains the **WWW-Authenticate** header with UMA-specific parameters: URL of the AM protecting the resource (**am_uri**) and the identifier of the Host at this AM (**host_id**).

Puma uses information from the **WWW-Authenticate** header to learn about the AM which can issue a RAT, and eventually an RPT. There are two conditions that have to be met by a Requester to be able to interact with this AM:

1. The application has to know the location of the AM's Authorisation API;
2. The application has to be registered as an OAuth 2.0 client with this AM.

The above steps can be achieved using (1) *discovery* and (2) *dynamic registration*, which were discussed in Sections 6.3.2.1 and 6.3.2.2 respectively. Further discussion assumes that these steps have been completed. A Requester has to complete the OAuth 2.0 authorisation, which was introduced in Section 6.3.2.3.

6.3.4.3 Obtaining Requester Access Token

Puma supports obtaining RAT tokens as following. Firstly, the framework creates a **RequestingParty** entity. This entity is the requester-equivalent of an **UMAUser** object (recall Section 6.3.3.1). The framework also creates **RequesterAccessToken** entity, which has a dichotomous role:

1. Contains a RAT for a specific AM;
2. Contains a set of RPTs for Hosts (internally called an *RPT Wallet*).

To obtain a RAT, a Requester uses a standard OAuth 2.0 flow. Puma supports generation of a URL to which a user is redirected. A user has to authorise an application for an AM and this

step was discussed in Section 6.3.2.3. In particular, when returning from an AM, a Requesting Party is redirected to a callback URL at a Requester. This callback has to be implemented as a handler because Puma does not provide a generic code, which could be included in a client application. Such handler extracts the authorisation code returned by an AM and exchanges it for a RAT. This RAT is used by a Requester to interact with the AM's *Authorisation API*. Importantly, a RAT is user specific and each Requesting Party has to be associated with their own token.

6.3.4.4 Obtaining Requester Permission Token

Accessing protected resources at Hosts requires that Requester presents a valid RPT. A Host evaluates RPTs, either locally or using an AM, and acts according to the UMA protocol. In case an RPT is missing in the request, then a Host provides Requester with information where such token can be obtained.

Puma Requester can check if an RPT is already present for the previously obtained RAT for a Requesting Party. If yes, then this RPT is included in requests to resources stored at a Host. If not, a Requester uses the AM's *Host Token* endpoint⁷ to get a new RPT.

RPT does not have any permissions associated when it is first issued to a Requester. Such RPT is issued without any policies being checked at an AM. A Requester associated a new RPT with the `RequesterAccessToken` entity. This RPT is then used in all further requests to protected resources at a Host. A simplified example code that implements the aforementioned behaviour of Puma is given in Listing 15.

```
1  rpt = Puma.Util.get_rpt_for_host_id(rp, am, host_id)
2  if not rpt: # There is no RPT for the host
3      rpt = Puma.Util.obtain_and_store_rpt_for_host_id(rp, am, host_id)
4  headers = { 'Authorization': 'Bearer ' + str(rpt) }
5  hx.request('GET', url, '', headers)
6  re = hx.getresponse()
```

Listing 15: Requester obtains an empty RPT and includes it in a request to a protected resource on a Host (simplified code) [124].

HTTP requests, which are equipped with an RPT, can result in one of the following responses from a Host:

1. HTTP 200 OK - a Host determined that an RPT carries enough permissions, and allowed

⁷The SMARTAM V2 implementation, which is presented in Chapter 7, provides a separate endpoint for obtaining new RPTs and a separate one for associating permissions with existing RPTs.

access; the response body contains the requested resource⁸.

2. HTTP 403 Forbidden - an RPT is valid but it does not have the required permissions. The response contains a **WWW-Authenticate** header with the included permission ticket. This ticket should be used to upgrade (i.e. assign new permissions) an RPT. New permissions can be assigned to an existing RPT or a new RPT can be issued.

The HTTP 403 Forbidden response is returned for RPTs that do not carry any permissions. Therefore, a Requester sends such an RPT to the AM's *Permission Request* endpoint. This endpoint serves the purpose of assigning new permissions to RPTs. An AM either assigns permissions to the RPT (or issues a fresh RPT with newly assigned permissions) or responds with a claims-requested document. In the latter situation, a Requester can engage in the authorisation phase.

A simplified example code, which is used to process a response from a Host and is used to interact with the AM's *Permission Request* endpoint, is given in Listing 16.

```
1  if re.status == 403:
2      www_auth = re.getheader('WWW-Authenticate')
3      ticket = Puma.Util.ticket_from_www_auth(www_auth)
4      claims_requested = Puma.UMA.send_ticket_to_claims_endpoint(rat,
5                                                                ticket, www_auth)
6      for claim in claims_requested:
7          if claim['claim_type'] == 'redirect_required':
8              self.redirect(claim['claim_value'])
```

Listing 16: Requester processing responses from Host and AM with attached claims information (simplified code) [124].

As presented in line 7 of Listing 16, a claims-requested document is a `claims_requested` object, which is returned by an AM. This object contains an array of claims that a Requester needs to provide in order to satisfy an access control policy.

Puma Requester supports integration with SMARTAM V2 implementation only (this AM is demonstrated in Section 7.4). In this AM, there is only one type of claims that a Requester can encounter: `redirect_required`. When such a claim is received by a Requester, then a Requesting Party is redirected to the location contained in the claim. As shown in Section 7.4, SMARTAM V2 leverages this redirect to validate identity and other attributes of a Requesting Party.

⁸Importantly, the UMA specification has been recently changed to allow different responses from a Host to support arbitrary APIs and not only HTTP GET requests.

After the necessary claims are provided to an AM, a Requesting Party is redirected back to Requester's callback handler. Puma does not provide a generic handler which could be used by client applications. Therefore, a custom implementation is used by the CareerMonster application.

This handler, apart from acting as a regular OAuth 2.0 callback handler, also performs upgrades of RPTs. This callback handles four different cases, which are determined by the presence of a specific parameter in the request to this handler:

1. **code** - standard OAuth 2.0 flow, where authorisation code is returned to the Requester. This code can be later exchanged for a RAT. This parameter is used solely for RATs and not for RPTs, as discussed in Section 6.3.4.3.
2. **x-oauth_access_granted** - Post-claims redirect, which provides information to the Requester whether access was granted (i.e. if necessary permissions can be associated with an RPT).
3. **x-oauth_access_req** - Post-claims redirect, which provides information to the Requester whether access to a resource was requested at an AM.
4. **x-oidc_claims_submitted** - Post-claims redirect, where a Requesting Party has successfully submitted necessary claims to an AM, allowing the Requester to continue access to a protected resource.

If the callback handler finds the **x-oauth_access_granted** parameter in the HTTP request, then it checks if this parameter has the **true** value. The presence of this value means that a user has authorised Requester to access their protected resources on a Host. This flow is used in the *Person-to-Self* sharing scenario, where the Requesting Party is the same as the Authorising User.

On the other hand, if the handler finds the **x-oauth_access_req** parameter set to **true** in the HTTP request, this means that a request for access has been submitted to an AM. This situation can occur when an access control policy has not yet been created for a particular Requesting Party who seeks access to a protected resource. This flow is used in the *Person-to-Person* or *Person-to-Service* sharing scenarios. Requester can present a custom UI and inform a Requesting Party that a request for access has been submitted successfully.

Furthermore, the AM implementation, which is demonstrated in Chapter 7, supports OpenID Connect claims. The callback handler can process the **x-oidc_claims_submitted** parameter set to **true** in the HTTP request to this handler. Such parameter occurs in *Person-to-Person* sharing scenarios, where a Requesting Party successfully submitted necessary OIDC claims to an AM.

In cases where either `x-oauth_access_granted` or `x-oidc_claims_submitted` parameters are returned with their value set to `true`, then an RPT is considered to bear new permissions and a Requester should try to access a resource again. The access request to a protected resource should return HTTP 200 OK. The UMA protocol has been later adapted to allow arbitrary HTTP return codes to be returned by host applications and this change was dictated by the need to provide the UMA proposal for protection of arbitrary Web APIs (and not only to access resources using HTTP GET requests).

Importantly, the aforementioned parameters, i.e. `x-oauth_access_granted`, `x-oauth_access_req`, and `x-oidc_claims_status`, which are handled by the callback handler, have been first proposed in the Puma framework. These parameters are not part of the UMA protocol specification. These parameters are used in the SMARTAM V2 implementation, allowing to achieve the functionality that is presented in Section 7.4.6 of Chapter 7.

6.3.5 Limitations

Despite the fact that Puma has been developed after the completion of the UMA/j framework, it still contains certain shortcomings.

Firstly, Puma only provides functionality for host and requester applications. Unlike UMA/j, it does not support development of Authorisation Managers. However, such functionality was considered unnecessary during the framework development. The goal was to create a framework, which would allow development of new applications that would integrate with SMARTAM V2 rather than to support development of additional AM applications.

Puma was implemented to support integration with SMARTAM V2. Therefore, it only supports claims through redirects (which allow a Requesting Party to provide either self-asserted or third-party asserted claims). As discussed in Section 6.3.4.4, there is only one type of claims issued by SMARTAM V2 that a Requester can encounter: `redirect_required`. When such claim is received by a Requester, then a Requesting Party is redirected to the location contained in the claim. This differs from UMA/j, which provided support for self-asserted claims that would be conveyed by a Requester directly to an AM. Adding support for directly conveyed claims was not considered a priority during Puma development. However, such support is planned in further versions of this software.

Moreover, the implemented framework provides support for remote token validation only. Hosts have to refer to an AM to get information on permissions associated with an RPTs. This differs from the UMA/j proposal that provides some support for local token validation (see Section 6.2.5.3).

In terms of the implementation and deployment, the Puma framework has been mostly

tested with two applications (namely the aforementioned PDS and CareerMonster). Moreover, testing of the framework integration has been done primarily on the Google App Engine platform although Puma was designed and implemented to be independent from the underlying application framework. Testing of the framework itself is based on unit tests only while UMA/j contained integration tests as well.

The `Puma.Storage` module contains the `StorageInterface` that provides an abstraction layer for the persistent storage (and subsequently allows Puma to persist information in various datastores). Despite providing such an abstraction layer, testing of the Puma framework was limited only to using the Google's non-relational datastore. More thorough tests should be conducted to ensure that Puma can be easily used on other platforms.

Puma requires developers to provide configuration directly in the code. Such configuration is necessary for Host and Requester applications. For example, both application types require configuration that relates to dynamic registration. Host applications also define which resources are protected and which HTTP methods map to what actions on specific resources. Such application configuration should be provided in separate files and not within the code. This shortcoming of the framework is planned to be addressed in further versions of the developed framework.

The Requester part of the framework, as discussed in Section 6.3.4.3, currently requires developers to implement the callback URL as a Python handler. Puma itself does not provide a generic code that could be included in the application. Such generic code is planned to be provided in further versions of Puma.

6.4 Chapter Summary

This chapter presented User-Managed Access frameworks that allow applications to externalise their access control functionality and act as Hosts and Requesters, as defined by the UMA protocol. It first presented a Java implementation named UMA/j. It also presented a Python implementation named Puma. It discussed specific enhancements incorporated into both frameworks that would simplify the process of interactions between UMA-enabled applications. It also discussed existing limitations of both frameworks.

Chapter 7

Authorisation Managers

7.1 Introduction

This chapter demonstrates two distinct implementations of UMA Authorisation Managers. It also presents a conducted user study of the first implementation. This study allowed to identify numerous shortcomings that were addressed in the second implementation. The research path on AM development is visualised in Figure 7.1.

Both Authorisation Managers have been implemented as part of the SMART project [77]. Therefore, both implementations have been named SMART Authorisation Managers (SMARTAM) and are referred to in this thesis as SMARTAM V1 and SMARTAM V2. Where the version of the designed software is omitted in the discussion, this is done deliberately in order to highlight the functionality which exists in both implementations.

Both AMs have been integrated with various Web applications acting as Hosts and Re-

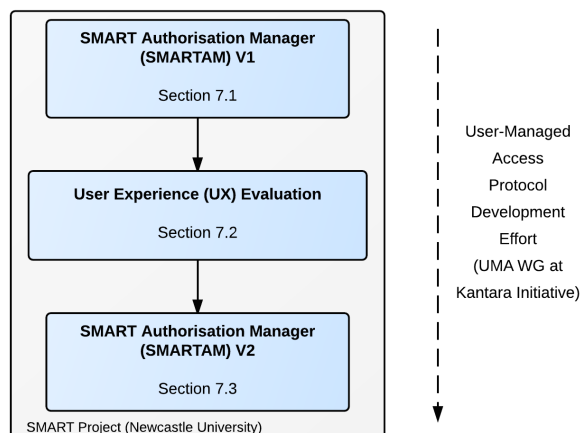


Figure 7.1: Research path on Authorisation Manager implementations.

questers. This proved that the UMA system can be successfully applied to different scenarios and can solve various use cases, such as those presented in [85] and [243], as well as that it can address the shortcomings identified in the scenario presented in Section 1.1.2. Such integration has been done using the implemented state-of-the-art UMA frameworks, which were discussed in Chapter 6.

Initially, Authorisation Managers were integrated with an online storage system and an online photo gallery service¹. These applications were named *Secure File System (SFS)* and *Gallerify.me* respectively. A Requester application, called *SmartFetch*, which acts as a client for both host applications, was also implemented. SmartFetch allows users to access files from SFS and photos from Gallerify.me.

Further focus was directed towards solving the "*Sharing Trustworthy Personal Data with Future Employers*" scenario, which was submitted to the UMA WG. A complete description of this scenario is given in [71]. An amended and simplified version of this scenario is presented in Section 1.1.2. Therefore, a Personal Data Store (PDS) application was developed. This application allows users to store personal information and academic records. The PDS application has been implemented as a demo of one the previously discussed online systems available at Newcastle University - S3P application [69]. A client application that can access UMA-protected resources at the PDS was also implemented. This application, named CareerMonster, mimics an online system where students can apply for advertised job positions. CareerMonster also allows prospective employers to request access to protected information from PDS and obtain such access based on approval from students. The flow of the improved scenario is showed in Appendix I.

The list of applications and information on their integration with the implemented Authorisation Managers is given in Table 7.1. This table also shows the framework, which was used for integration. By integrating several applications with Authorisation Managers, it was possible to design and implement very generic UMA software, which could be successfully applied to other Web applications as well.

During development of both Authorisation Managers, the UMA protocol has undergone various significant modifications. As such, this chapter restrains from providing revisions or versions of the protocol used by the implemented software. Importantly, the development efforts allowed to provide valuable feedback to the UMA WG and such feedback was incorporated into the core protocol specification.

This chapter also presents a conducted user study, which evaluated the User Interface of SMARTAM V1. Furthermore, this study also aimed to evaluate the User-Managed Access

¹ It was eventually decided not to continue development and support for the online gallery service. Therefore, this application is no longer accessible on the Internet.

Table 7.1: UMA-enabled Web applications developed during the course of research and their integration with developed Authorisation Managers.

Application	UMA role	Framework	SMARTAM V1	SMARTAM V2
Secure File System (SFS)	Host/Requester	UMA/j	✓	
Gallerify.me	Host/Requester	UMA/j	✓	✓ (with initial versions)
SmartFetch	Requester	UMA/j	✓	✓ (with initial versions)
Personal Data Store (PDS)	Host	Puma		✓
CareerMonster (PDS Client)	Requester	Puma		✓

protocol in general. Conclusions from this study were used to build a new version of the AM, named SMARTAM V2.

The remainder of this chapter is organised as follows. Section 7.2 presents an initial implementation of the UMA Authorisation Manager called SMARTAM V1. This AM was developed in the initial stage of UMA protocol development. In particular, it did not contain such features as resource or permission registration. Therefore, this AM is presented in limited form only, focusing on its provided functionality as well as its UX. This section also discusses its integration with the SFS and Gallerify.me applications.

Section 7.3 presents evaluation of the implemented SMARTAM V1. A user study was conducted to gather feedback on the UMA proposal and on the User Interface of the implemented AM. This study allowed to identify shortcomings in the initial AM implementation and to formulate requirements for further research.

Section 7.4 presents a second implementation of the UMA Authorisation Manager, named SMARTAM V2, which was based on the feedback of the conducted user study. This Authorisation Manager implemented a more recent version of the UMA protocol and provided various new features, such as resource or permission registration, among others. Both versions are compared in more details in Section 7.4.

7.2 SMART Authorisation Manager V1

SMARTAM V1 allows a user to compose access control policies and apply them to a set of resources hosted on different Web applications (Hosts). These applications delegate access control to this AM using the UMA protocol and do not have to provide any policy specification interfaces to end users. Instead, these applications refer to the AM to validate access requests to protected resources made by clients (Requesters) and only enforce access control decisions returned by this AM during the validation process. Requesters, on the other hand, can use the

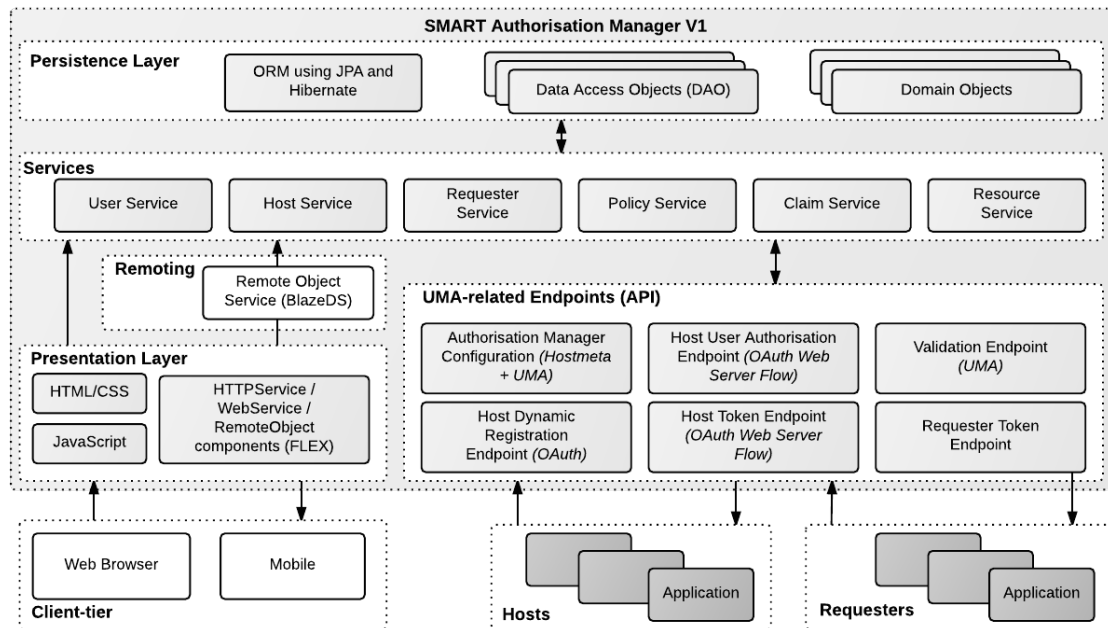


Figure 7.2: Architecture of SMART Authorisation Manager V1.

AM to obtain the necessary tokens to access protected resources on host applications. Hosts and Requesters are discussed in more details in Section 7.2.3.2 and Section 7.2.3.3 respectively.

SMARTAM V1 provides both a User Interface and a RESTful Web API. The UI can be used by a user to manage policies for their distributed resources stored at Hosts. In particular, SMARTAM V1 supports creating, updating, deleting and reading policies which are persisted in a datastore. The RESTful API is used for interactions with Hosts and Requesters according to the defined UMA protocol.

7.2.1 Architecture

SMARTAM V1 has been implemented as a multi-tier Web application, comprising of the following layers: *persistence layer*, *services layer*, *presentation layer*, and *API layer*. The architecture of this AM is visualised in Figure 7.2. The user is able to access the application with existing Web browsers or mobile devices². They use the provided UI to manage their registered applications, resources, access control policies and contacts.

Data in SMARTAM V1 is processed by different components of the services layer and different services are provided for each function. For example, the *User Service* is responsible for management of Authorising Users as well as Requesting Parties. The *Host Service* and *Requester*

²The implemented Authorisation Manager was also accessed on Microsoft Surface interactive tables. Very basic experiments were conducted on how to set access control policies by interacting with a screen. However, no results from these tests have been published.

Service manage client applications, which can interact with this AM. The *Policy Service* and *Claims Service* are responsible for policy evaluation and claims gathering functions. Similarly, the *Resource Service* is responsible for managing resources at the AM.

Services store and retrieve data from a SQL database. SMARTAM V1 does not interact with the database directly but it uses Object-Relational Mapping (ORM) engine for this purpose. Specific Data Access Objects (DAOs) are provided, which are used by services to get access to domain objects. For example, the *Policy Service* uses a *Policy DAO* to manage *Policy* domain objects in the persistent store.

Client applications interact with SMARTAM V1 using HTTP requests sent to the AM's RESTful Web API. This API exposes standard UMA endpoints. For host applications, it provides the following endpoints: *Authorisation Manager Configuration* endpoint (used in the discovery process), *Host Dynamic Registration* endpoint (dynamic registration), *Host User Authorisation* endpoint and *Host Token* endpoint (OAuth 2.0), and *Token Validation* endpoint (authorisation validation). Requester applications can use the *Requester Token* endpoint. These endpoints are discussed in details in Section 7.2.3.

The presentation layer has been developed as a *Rich Internet Application (RIA)* [174]. Therefore, the UI functionality runs within the user's Web browser and resembles a desktop application. Communication between the UI and backend Java services uses BlazeDS. In particular, specific actions conducted by a user at the UI trigger invocations of remote Java methods, while server responses are translated for representation at the UI (e.g. saving a policy communicates required information to the *Policy Service* on the server, which can return a success response that is displayed to a user).

7.2.2 Security

A user has to authenticate in order to access the UI of the SMARTAM V1. This AM does not support any federated authentication protocols. Instead, the user needs to register manually and the sign in process is based on username/password authentication.

Security of the API is provided with username/password authentication, HTTP Basic Authentication and the OAuth 2.0 protocol. API endpoints that are required for user interaction, such as the *User Authorisation* endpoint, require username/password authentication. On the other hand, endpoints that provide functionality to client applications are protected with OAuth 2.0. In particular, the *Validation* endpoint requires an OAuth 2.0 token to be present in the HTTP request. Furthermore, the *Requester Token* endpoint requires credentials to be included in the **Authorization** header of the HTTP request. Importantly, all communication between different parties and SMARTAM V1 is done using the HTTP protocol with TLS/SSL.

7.2.3 Support for User-Managed Access

User-Managed Access functionality is exposed through an API. This API is composed of all the necessary endpoints to achieve the required functionality of centralised authorisation.

This API allows Web and mobile applications to discover available AM endpoints. Discovery is done through the *Authorisation Manager Configuration* endpoint, which returns an XRD-formatted document that lists endpoints, among other configuration options.

The *Host Dynamic Registration* endpoint allows applications to register themselves as clients dynamically, using the protocol discussed in [237]. Registration allows applications to be provisioned with the necessary OAuth 2.0 credentials, i.e. $\{client_id, client_secret\}$, that are later used during communication with AM. It is important to note, that SMARTAM V1 only requires registration of host applications and not requester applications (i.e. Requesters act as anonymous clients without any client credentials).

The UMA API also exposes further additional endpoints for Hosts and Requesters. These endpoints use the components provided by the services layer. For example, obtaining authorisation by Requester using the *Requester Token* endpoint, requires this endpoint to interact with the *Policy Service* as well as *Claim Service*.

SMARTAM V1 has been developed according to one of the early revisions of the UMA protocol. Therefore, it did not contain all the features that were described in Chapter 5. In particular, this AM differs in the following aspects from the described UMA proposal:

1. AM provisions requester applications with single tokens;
2. Resource registration is not supported;
3. Permission registration is not supported.

The missing resource registration requires users to register resources manually at the AM with the provided UI. On the other hand, the missing permission registration requires a simplified UMA flow when a Requester attempts to access a protected resource. Furthermore, SMARTAM V1 issues **Host Access Tokens (HAT)** to host applications and **Authorisation Tokens (AT)** to requester applications (recall additional RPT tokens discussed in Chapter 5). These differences are discussed in further sections.

7.2.3.1 Authorisation tokens

SMARTAM V1 supports the following tokens:

1. **Host Access Token** - this token is used by the host application to communicate with the AM when validating **Authorisation Tokens** received from Requesters.

2. **Authorisation Token** - this token is obtained by the requester application from AM's *Requester Token* endpoint. The Requester needs to exchange credentials of the Requesting Party (as well as optional claims) in order to be provisioned with this token.

Tokens in SMARTAM V1 are bearer tokens, which are opaque to host and requester applications. These tokens have similar characteristics to single-sign-on (SSO) cookies used in Web browsers (e.g. after the user establishes a session with a Web system). Tokens in SMARTAM V1 are at least 128 bits.

7.2.3.2 Support for host applications

A user can authorise their host applications at AM and delegate their access control functionality from these applications. The OAuth 2.0 Web Server Flow (Authorisation Code Grant) has been implemented for this purpose [202]. In particular, SMARTAM V1 exposes standard OAuth 2.0 endpoints for this purpose, i.e. *User Authorisation* and *Token* endpoints. When provisioned with an access token by the AM, these host applications are able to validate received Authorisation Tokens from Requesters using the available *Validation* endpoint.

The *Validation* endpoint supports Hosts with evaluating access requests to protected resources. This endpoint is slightly different from the one discussed in Section 5.4.5. In particular, host applications send an HTTP POST request with a JSON-formatted body, containing the `realm`, `method`, and `token` fields of type `String`. These fields represent the URL of the resource, HTTP access method used by the Requester, and the Authorisation Token received from the Requester respectively.

The Host uses its own HAT in order to get access to the *Validation* endpoint. Importantly, the Host can only validate access requests to resources stored on this particular host. The AM responds with a simple access control decision, provided as a value (either `permit` or `deny`) of the `decision` field in JSON-formatted object. Policy evaluation which leads to such decision is discussed in Section 7.2.5.

7.2.3.3 Support for requester applications

Requester applications, unlike Hosts, are not required to be registered as OAuth 2.0 clients with SMARTAM V1 and these applications can act as anonymous clients (i.e. these applications are not provisioned with their own $\{client_id, client_secret\}$ pair). Requesters interact with the provided *Requester Token* endpoint at the AM on behalf of the user. This endpoint provides functionality according to OAuth 2.0 Username and Password flow (Resource Owner Password Credentials) [202] where credentials are provided using the HTTP Basic Authentication scheme

[192]. Importantly, Requesters do not use any of their own credentials but only provide username and password on behalf of the Requesting Party.

Requesting Parties, before accessing protected resources, have to meet access control policies defined at the AM. This may require Requesting Parties to use the username and password that were provisioned to them by an Authorising User. A Requesting Party provides their credentials at the requester application, which sends these credentials to the AM on behalf of that user. If the provided credentials are sufficient to meet an access control policy (or policies) then the Requester is provisioned with the Authorisation Token for a specific protected resource at the Host. In case access control policies contained any claims then a Requesting Party may need to provide self-asserted claims before the Authorisation Token is issued. The Requester collects these claims from the Requesting Party and sends them to the AM on behalf of that user. The process of collecting claims by Requesters is discussed in Section 7.2.5.1.

Importantly, the information (either Requesting Party's credentials or self-asserted claims) are always sent on behalf of the user and the Requester itself does not send any credentials of its own (i.e. it acts as an anonymous client to the AM). Along with the Requesting Party's credentials or claims, the Requester also passes information, in the HTTP POST request, regarding URL of the protected resource, and the method that it wishes to use on that resource. This information is passed as `uri` and `method` fields of a JSON-message.

It is important to note that the credentials that are submitted by Requesting Parties at requester applications are not used in any other sign in processes. In particular, these credentials are only issued by Authorising Users for Requesting Parties to access a specific set of protected resources. For example, an Authorising User generates such credentials at the AM and provides them to a Requesting Party out of band for use at this specific AM. These credentials are then exchanged them for an Authorisation Token that is scoped for a specific set of resources.

SMARTAM V1 is authentication agnostic and does not define how a Requesting Party should authenticate to the AM. A Requesting Party can present its credentials, such as username and password, or use a SAML assertion to authenticate. SMARTAM V1 focuses primarily on the HTTP Basic Authentication scheme [192] to exchange username and password for an Authorisation Token. Such authentication method is available in OAuth 2.0 specification but requires the use of transport-level security. As already discussed, all endpoints exposed by SMARTAM V1 are available over HTTP with SSL/TLS. Basic support for SAML assertions [151] has been also implemented but it has not been tested.

It is important to note that the scope of access for an Authorisation Token is specified by an Authorising User in the access control policy (see Section 7.2.4). The initially proposed solution contains shortcomings and these relate to credential management (e.g. provisioning credentials

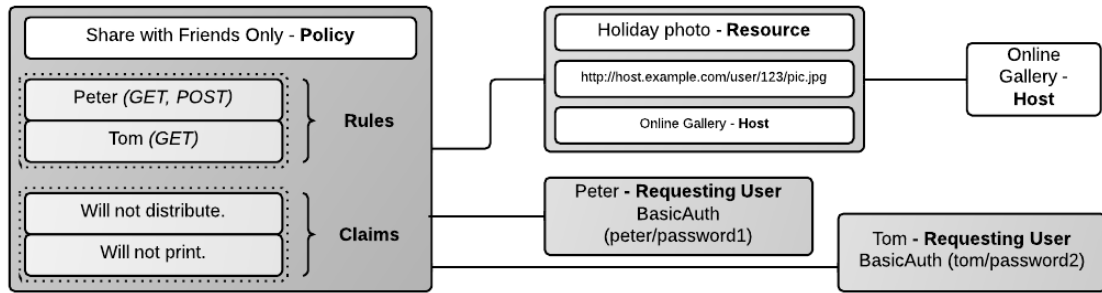


Figure 7.3: Example implementation of an access control policy.

to Requesting Parties, managing multiple different credentials by Requesting Parties, password resets, etc.). These shortcomings are discussed in more details in Section 7.2.9. Redesigned interactions between Requesters and AM are shown on the example of a second AM implementation in Section 7.4.

7.2.4 Policy Model

Authorisation Tokens are issued to Requesters based on user defined policies and the AM allows users to create such policies and apply these policies to resources stored at multiple Hosts, which are registered with SMARTAM V1. Policies can be created once and can be reused across different resources. Moreover, multiple policies can be linked with a single resource.

The implemented policy model comprises of two sets: *Rules* and *Claims*. A single rule defines a set of Requesting Parties and a set of access rights that these users possess. Access rights are expressed in the form of HTTP methods that can be executed on a resource (e.g. user can specify only **GET** or only **POST** requests). A single claim defines a statement that Requesting Parties must assert to the AM. For example, Requesting Party must assert that they are over 18 years old or that they will not print the photos that they are accessing. Claims are self-asserted and the AM implementation only collects user confirmation to those statements. An example of a policy is given in Figure 7.3.

A workflow for creating a policy and associating this policy with a resource at the implemented Authorisation Manager comprises of the following steps:

1. User creates a new Access Control policy element;
2. User defines a new Rule to be added to the AC policy;
3. User associates an existing Claim with the AC policy;
4. User links a Resources with the AC policy.

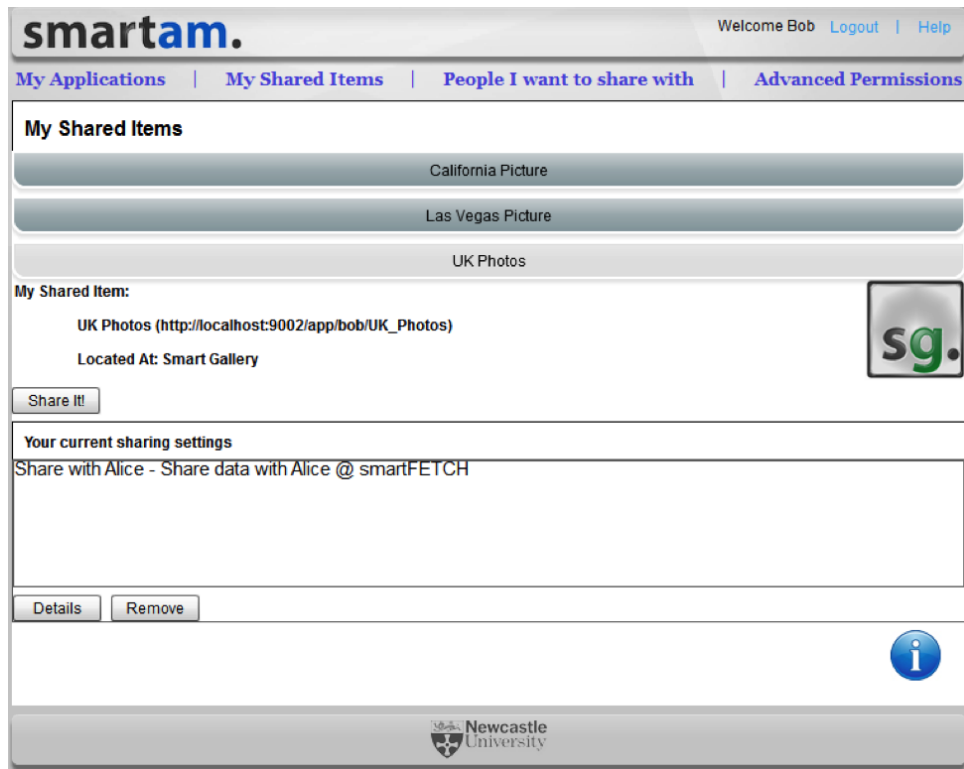


Figure 7.4: Example of a resource linked with an access control policy.

The UI of SMARTAM V1, which supports this workflow, is shown in Section 7.2.6. Once a policy is created and linked with a resource, it can then be viewed by an Authorising User through the provided UI of the AM. Importantly, resources registered at the AM can be linked with a single or multiple policies (Figure 7.4).

7.2.5 Policy Evaluation

Before accessing a resource, a Requester has to be in a possession of an Authorisation Token. Therefore, it uses the *Requester Token* endpoint at the AM to obtain such token. As discussed in Section 7.2.1, a Requester issues the HTTP POST request with Requesting Party credentials as well as the URI of a resource and the HTTP method of access to this resource. Based on the information contained in the request, the AM can search for applicable policies, which were specified by an Authorising User. These policies are used for the access control process.

Access control policies in SMARTAM V1 can be complex and may require users to authenticate themselves or negotiate authorisation. In the first case, policies require that the Requesting Party authenticates to the AM before access can be granted. The AM communicates the need for authentication in reply to the Requester's HTTP request issued to *Requester Token* end-

point. This reply contains the required authentication type (SMARTAM V1 supports HTTP Basic Authentication).

In the latter case, there can be a more complex process of negotiating access to a resource based on claims provided by Requesting Parties. This process involves sending a *claims requested* document to the Requester and evaluating the *claims* response.

In SMARTAM V1, there can be multiple different policies associated with a resource or a group of resources. With multiple policies in place, it is common for policy conflict resolution methods to be used. This AM currently supports only positive statements in access control policies (see Section 2.3.1). Therefore, it does not require such methods to be adopted.

Evaluating access requests against access control policies is currently performed by a custom-built policy evaluation engine, which checks whether the Requesting Party should be provisioned with the Authorisation Token. Following is a short description of how policies are evaluated:

1. If a policy has only *Rules* then Requesting Parties must authenticate to the AM to prove their identity. Then policy evaluation engine checks if the requested access type matches the one specified in the applicable rule;
2. If a policy has only *Claims* then all Requesting Parties must provide these claims. Access type is restricted to defined types in the policy. Authorisation phase may require the Requester to exchange multiple claims documents with the AM (see Listing 20);
3. If a policy has both *Rules* and *Claims* then Requesting Parties must authenticate to the AM and also submit claims for evaluation by the implemented policy engine. Only then access to a resource can be granted.

7.2.5.1 Claims

The Claims 2.0 proposal [247] has been used and adapted in SMARTAM V1. Claims 2.0 defines a JSON-based format for expressing claims and requests for claims. The AM informs the Requester about the list of claims that must be submitted using the *claims requested* document included in the body of the HTTP response (see Listing 18 and Listing 17).

```
1 HTTP/1.1 401 Unauthorized
2 Content-Type: application/x-www-form-urlencoded
3
4 claims_requested={...claims_requested_document...}
```

Listing 17: Response from SMARTAM V1 containing information about required list of claims.

```
1 { "http://c2.io/claims-requested": [  
2   { "type": "https://am.example.com/claims/confirmation",  
3     "description": "You must acknowledge to be > 18 years old." } ]  
4 }
```

Listing 18: Example of a claims requested document issued by SMARTAM V1.

When the Requester receives a *claims requested* document, it collects the claims from the Requesting Party and sends the *claims* document back to the AM. Such collection may require the Requesting Party to provide consent to specific terms or include additional information that will be presented to the AM. Importantly, the Requester can exchange multiple claims documents with the AM before authorisation can be granted (recall Figure 5.9 - C).

In UMA, claims can be either *self-asserted* or *third-party asserted*. The example presented in Listing 19 shows a self-asserted claim that the Requesting Party is over 18 years old. Third-party asserted claims additionally contain the issuer of the claim as well as the signature of the claim [247]. SMARTAM V1 provides support for self-asserted claims only, which is one of this AM's limitations (see Section 7.2.9). Third-party asserted claims are discussed in Section 7.4. Claims received from the Requester can be recorded by the AM for accountability purposes before issuing authorisation. Importantly, the AM may negotiate the necessary set of claims based on the implemented logic and during the authorisation phase. A simplified code used to evaluate submitted claims before issuing the Authorisation Token is shown in Listing 20.

```
1 { "http://c2.io/claims": [  
2   { "type": "https://am.example.com/claims/confirmation",  
3     "description": "You must acknowledge to be > 18 years old.",  
4     "value": "YES" } ]  
5 }
```

Listing 19: Example of a claims document with a self-asserted claim sent by a Requester to SMARTAM V1.

7.2.5.2 Authorisations

SMARTAM V1 can grant an authorisation by issuing an Authorisation Token that is bound to the access request and cannot be used to access other resources protected by this particular AM (i.e. the token is scoped for a particular access request). The AM stores such token in its persistent store, along with information regarding the Host, protected resource, HTTP method,

```
1 List<Claim> claimList =
2     claimDao.findClaimsForRule(rule.getPk().getRule().getId());
3 List<Claim> necessaryClaims = new ArrayList<Claim>();
4
5 if (!claimList.isEmpty()) {
6     List<ClaimSubmitted> submittedClaims =
7         JSONClaimsProcessor.parseClaims(claims);
8     for (Claim claim : claimList) {
9         boolean claim_satisfied = false;
10        for (ClaimSubmitted submittedClaim : submittedClaims) {
11            if (ClaimEngineImpl.evaluate(submittedClaim, claim)) {
12                claim_satisfied = true;
13                break; }
14        }
15        if (!claim_satisfied) {
16            log.debug("Adding claim.");
17            necessaryClaims.add(claim); }
18    }
19 }
```

Listing 20: Evaluating submitted claims by the claims engine at SMARTAM V1 (simplified code).

requesting party, as well as the timestamp when the token was issued. This information is stored for accountability and audit purposes. It is also used to determine whether a Host should grant access to a particular resource (see next section).

```
1 HTTP/1.1 200 OK
2 Content-Type: application/x-www-form-urlencoded
3
4 authorisation_token=07cJwX2W352t9x9T
5 &issued=1360587000
6 &expires_in=3600
```

Listing 21: Authorisation token response sent by AM to Requester.

The Authorisation Token is passed back to the Requester using the `authorisation_token` parameter. The response from the AM also contains the timestamp when the token was issued and its lifetime (using `issued` and `expires_in` parameters, respectively). This token is used by the Requester when issuing access requests to protected resources at the Host.

The initial UMAC proposal, as discussed in [241; 242], was based on the access control pull model that did not require an Authorisation Token to be acquired from an AM and was transparent for a Requester. The additional step of obtaining such a token based on the OAuth WRAP proposal [119] has been introduced. A similar approach is also adopted by the UMA

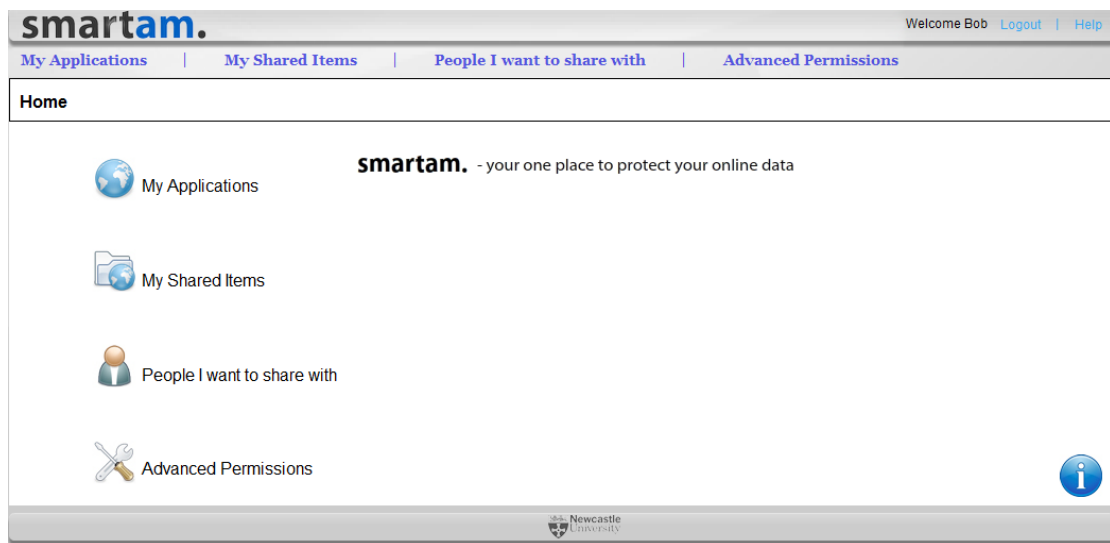


Figure 7.5: Main page of the SMARTAM V1 UI.

protocol [297] that SMARTAM V1 implements. UMA has based its protocol on WRAP at some point as well (UMA was initially based on OAuth 1.0a). Originally in UMA, the Requester would not obtain a token from an AM but would rather establish an authorisation state for a particular scope on a particular Host. This state would then be checked by a Host when querying an AM for an access control decision.

7.2.6 User Interface

SMARTAM V1 provides a User Interface that allows Authorising Users to manage their registered applications, resources, policies, and contacts. The layout of this UI, as presented in Figure 7.5, is divided into four major views. These views are available from a horizontal navigation bar and directly from the main page of SMARTAM V1:

1. My applications;
2. My shared items;
3. People I want to share with;
4. Advanced permissions.

My applications category displays a list of host applications that are registered at the AM (Figure 7.6). These applications have to be dynamically registered using the previously mentioned *Host Dynamic Registration* endpoint. This view provides a name, URL and icon for each



Figure 7.6: View displaying registered applications at AM.

listed application. Furthermore, each host application is associated with a *Share Item* button, which allows users to manually register resources for protection.

The *My shared items* view displays the pivotal part of the application, i.e. the list of protected resources (Figure 7.7). This list is shown with use of an accordion component. Each item on the list is provided with a brief description, including the name of the registered resource and the URL of the resource at the host application. Furthermore, underneath the resource description there is a list of currently applied access control policies. At the UI level, these policies are referred to as **Sharing Settings**. Each policy has a user defined name. By selecting a particular policy from the list and clicking on the *Details* button, an overlay is displayed with a list of Requesting Parties associated with this policy that have been granted permission to access this resource. Additionally, an Authorising User can view the list of applied restrictions. As discussed in Section 7.3, Requesting Parties are stored as *Rules*, while restrictions are stored as *Claims* in access control policies.

The *People I want to share with* view shows a list of registered Requesting Parties at this AM for this particular Authorising User (Figure 7.8). Each Requesting Party represents a set of credentials that have to be provisioned out of band.

The *Advanced permissions* view shows a list of existing claims, which can be used when composing access control policies (7.9). The user can add new claims as necessary.

In order to add a new access control policy, the user is required to click on the *Share It* button available at the resource view. The button triggers a sharing options overlay, which allows users to associate access control policies for a resource. If the available access control

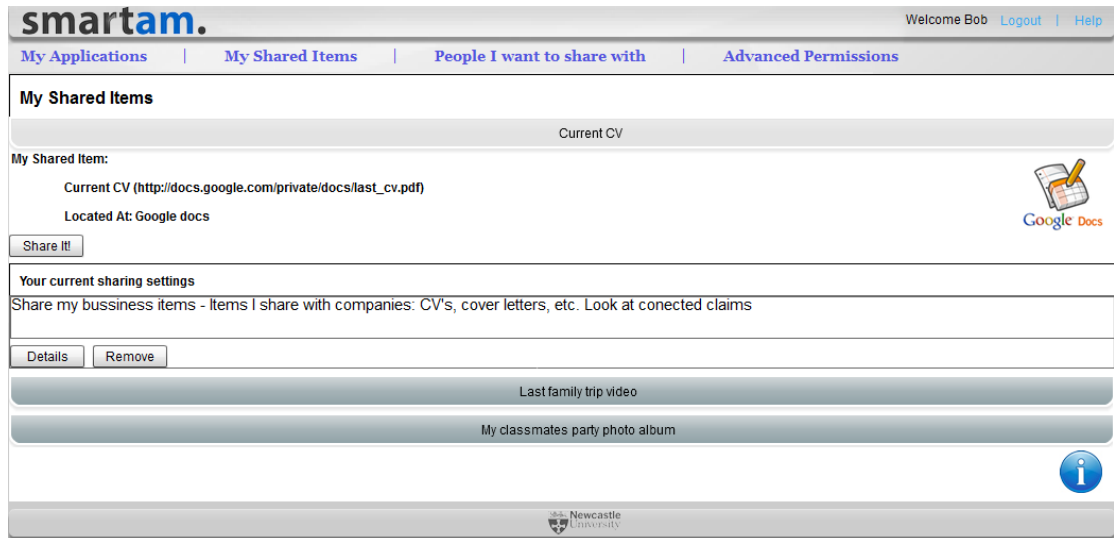


Figure 7.7: View displaying registered resources at AM.

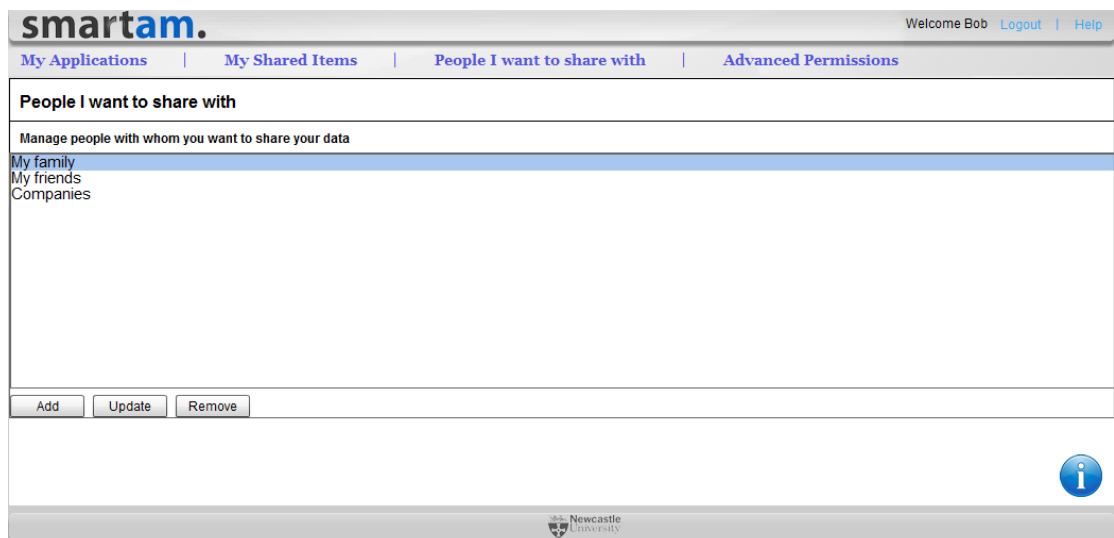


Figure 7.8: View displaying a list of Requesting Parties created by an Authorising User.

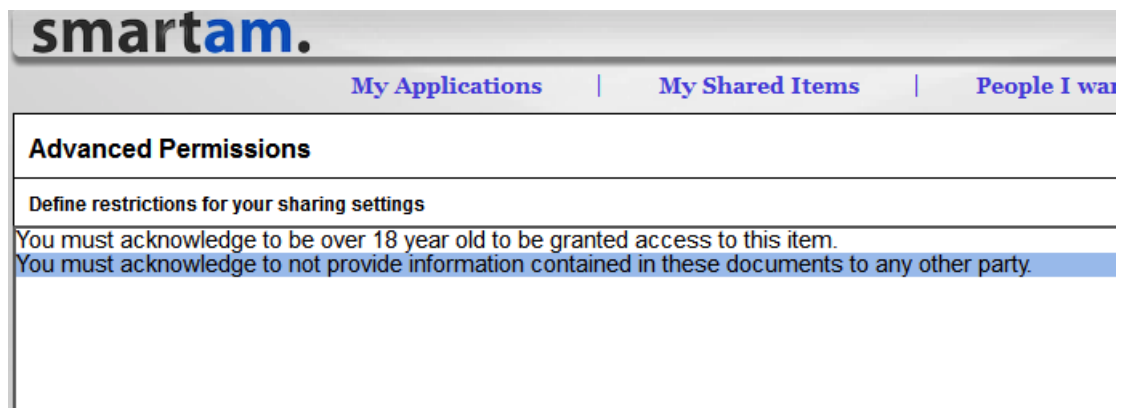


Figure 7.9: View displaying list of available restrictions (claims).

policies are insufficient for a particular resource then new policies can be created. Users can create rules for new policies and can also apply restrictions for these policies.

When discussing the policy creation process further in this section, it is assumed that the user has already registered a set of Requesting Parties and provisioned these parties with credentials. This can be done out of band and is not a part of the UMA protocol or the presented AM implementation. It is also assumed that the user has created a set of Claims. SMARTAM V1 allows a user to create new claims using the provided UI and this is discussed further in this section. Registered Requesting Parties and claims can then be used in multiple policies at the AM. The necessary steps of creating a new access control policy for a resource are presented in Figures 7.10, 7.11, 7.12, and 7.13.

Figure 7.10 shows how a policy is created using the implemented AM. The user has to provide a name of the policy as well as its description. Such policy is then associated with roles and/or with claims. A single rule requires a name, an access method to which it refers to (this access method is an HTTP verb), as well as the set of Requesting Parties. As discussed earlier in this section, each Requesting Party represents a set of credentials that have to be provisioned by the user out of band. For example, "*Alice @ smartFETCH*" represents credentials provisioned to a user *Alice* that uses an application called *smartFETCH* to access protected resources (refer to Figure 7.11). Credentials are discussed in Section 7.2.7.

Policies can be also associated with claims. Each individual claim in the AM contains a *name* and a *statement*. The statement is a simple string that the user, who composes a policy, will require from the requesting user to be confirmed before access to a resource can be granted. As depicted in Figure 7.12, the user can associate a set of claims with a policy and the AM implements a drag and drop UI for this purpose (the user has to drag a policy from the set of "*Available Claims*" to the set of "*Selected Claims*"). When rules and claims have been defined

The screenshot shows a web application window titled "Holiday Video Clips". Inside, there is a "Policy details" form. The form has two input fields: "Policy name:" with the value "Share with Friends" and "Policy description:" with the value "Only friends @ smartFetch". Below these fields is a section titled "Rules" which contains a sub-section "Sharing rules:" followed by a large empty text area. At the bottom of the "Rules" section are three buttons: "Add Rule", "Update Rule", and "Delete Rule". Below the "Rules" section is a section titled "Claims" which is currently empty. At the very bottom of the form are two buttons: "Cancel" and "Save".

Figure 7.10: View for creating new access control policies.

for a policy, the user can associate this policy with a resource and SMARTAM V1 provides a drag and drop UI for this purpose (Figure 7.13). A composed policy can be later used for other resources as well.

7.2.7 Integration with Applications

SMARTAM V1 has been integrated with two prototype host applications - an online Secure File System and an Online Gallery Service. Both applications are introduced in Section 4.3.3 of Chapter 5. The first one is an online file system accessible over the Web where the user can upload arbitrary files and create an arbitrary directory structure. The second application allows a user to upload photos and create photo albums. In addition, it allows the user to edit their photos (resize, rotate, crop, etc.). Thus, this application also acts as a Web-based photo editing tool.

The previously implemented requester application has been adapted to support the UMA protocol. The application interacts with two host applications through their RESTful APIs and implements the logic of the UMA protocol. Simplified code of the implemented Requester is

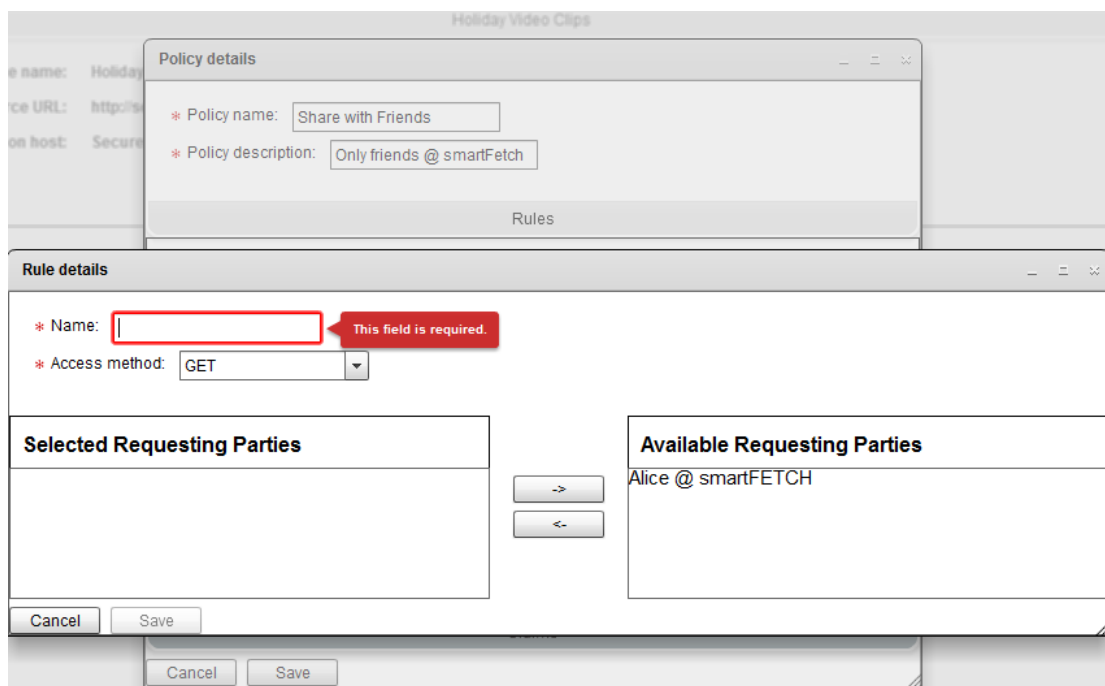


Figure 7.11: View for creating new rules to be added to an access control policy.

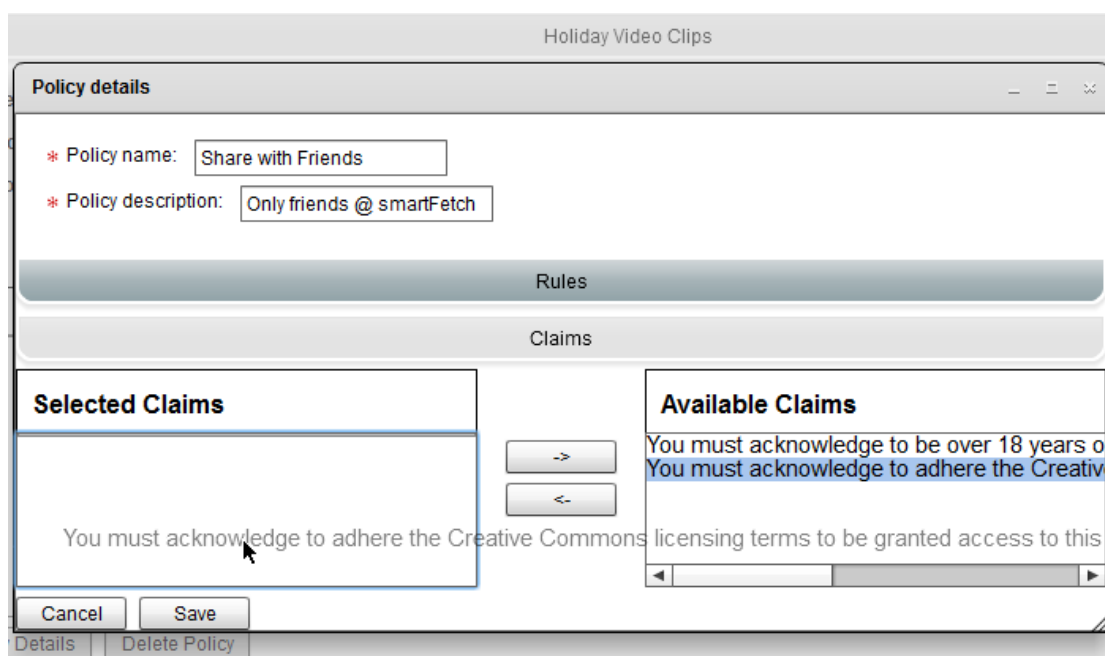


Figure 7.12: View for associating existing restrictions (claims) with an access control policy (drag and drop).

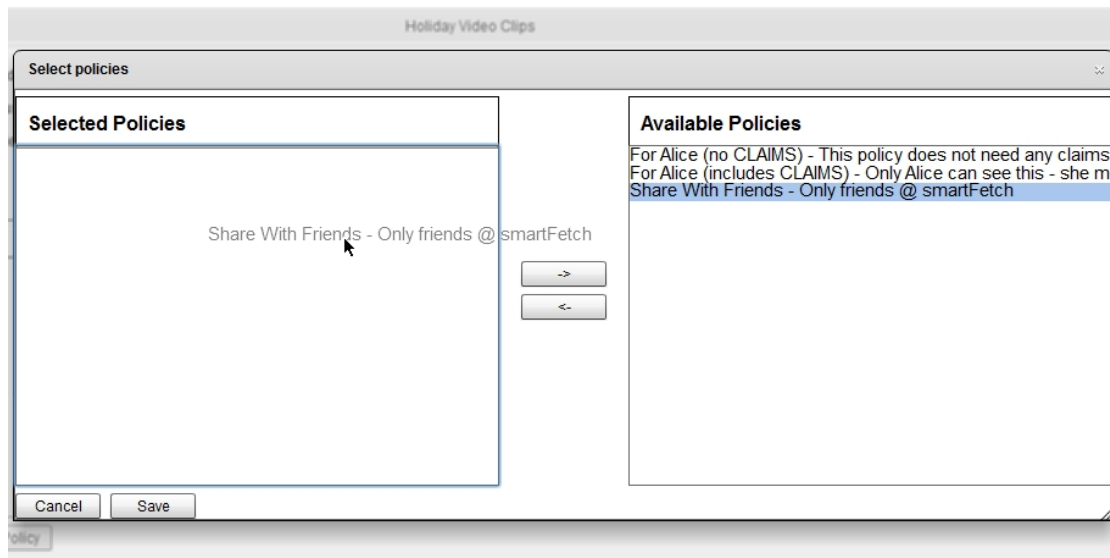


Figure 7.13: View for associating an access control policy with a resource (drag and drop).

presented in Listing 22.

When a Requester tries to access an UMA-protected resource and this Requester does not yet have the Authorisation Token, then it will require the Requesting Party to act. The Requester can detect the AM that protects the resource. It then allows the Authorisation Token to be acquired from this AM for different types of policies, including ones composed of rules and claims. Importantly, it can authenticate Requesting Parties against SMARTAM V1 and can participate in the authorisation phase.

Authentication of Requesting Parties has been implemented using OAuth 2.0 Username and Password flow where credentials are passed according to HTTP Basic Authentication scheme [192] (recall Section 7.2.3.3). An example of obtaining credentials from a Requesting Party is depicted in Figure 7.14. Furthermore, the authorisation phase allows the Requester to collect claims from the Requesting Party and present those claims to the AM in order to obtain the actual authorisation for a resource (or a set of resources). SMARTAM V1 supports self-asserted claims and allows the Requesting Party to confirm terms imposed by the user that owns the protected resource. The requesting user may confirm all or only selected claims and if these claims are insufficient then the Requester can present additional ones, depending on the response from the AM (recall Listing 20). For example, the user could confirm only one of the presented claims and this would not necessarily result in access being denied. Instead, the AM could then present a different set of claims to the user based on the implemented access control policy. This process could be repetitive up to a point where the AM can gather the set of claims sufficient to grant access to a resource. However, SMARTAM V1 currently does not allow such flexible policies to

```

1  public String getResource(String resourceUrl, String token) {
2
3      GetMethod hostResourceMethod = new GetMethod(resourceUrl);
4      Header header = null;
5      if (accessToken != null) {
6          header =
7              new Header("Authorization", "token=\"" + token + "\"");
8      }
9      hostResourceMethod.setRequestHeader(header);
10     try {
11         hostResourceMethod.setDoAuthentication(false);
12         client.executeMethod(hostResourceMethod);
13
14         if (hostResourceMethod.getStatusCode() ==
15             HttpServletResponse.SC_UNAUTHORIZED) {
16             // ...try to acquire authorisation token...
17             // ...authentication (optional)
18             // ...claims submission (optional)
19         } else if (hostResourceMethod.getStatusCode() ==
20             HttpServletResponse.SC_OK) {
21             // ...get the resource from the response...
22         }
23     } catch (Exception e) {
24         // ...
25     } finally {
26         hostResourceMethod.releaseConnection();
27     }
28     // ...
29 }

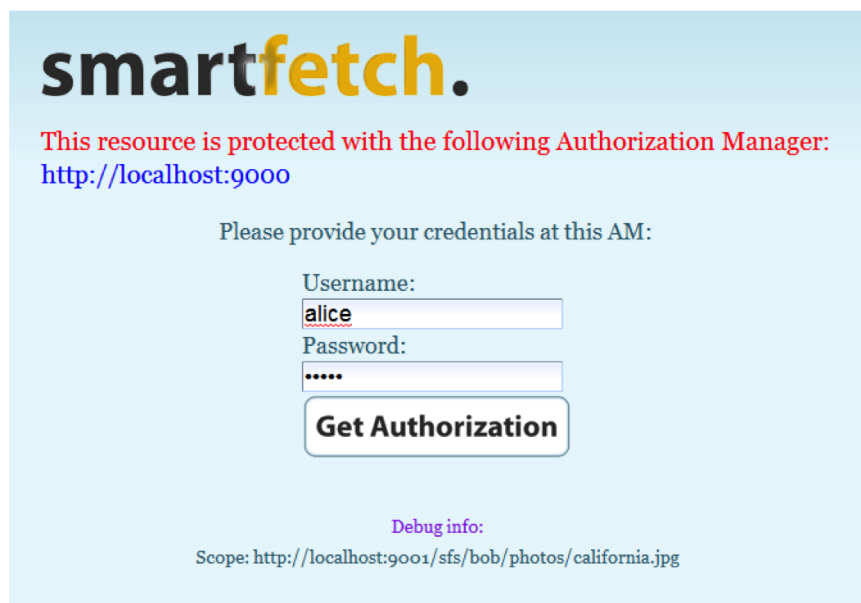
```

Listing 22: Example implementation of an UMA Requester client code (simplified).

be composed by the user (this limitation is discussed in Section 7.2.9).

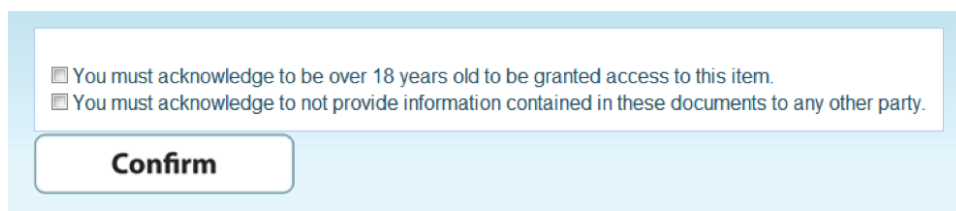
The requester application provides a new user experience to the end-user that acts as a Requesting Party by showing an additional screen (or a set of screens) before access to a resource can be granted. This is dictated by the requirement of authentication of the Requesting Party to the AM as well as by the necessity to gather claims that have to be provided to AM prior to obtaining the authorisation token. Such UX in the presented solution is similar to existing ones, such as OAuth, that require the user to authenticate and authorise an application before accessing a resource. The presented UX has been updated and refined to be more aligned with the UX proposed by OAuth that is based on redirects and is already widely accepted on the Web. This new user experience of accessing a resource is presented in Section 7.4.6.3.

Host and request applications have been implemented in Java and deployed to the Google App Engine platform [21]. These applications make use of the rich set of APIs provided by the



The image shows a web interface for 'smartfetch.'. It has a light blue background. At the top, the text 'smartfetch.' is displayed in a large, bold, black font, with 'smart' in black and 'fetch.' in yellow. Below this, a red line of text states: 'This resource is protected with the following Authorization Manager:'. Underneath, a blue link shows the URL 'http://localhost:9000'. A black text prompt asks the user to 'Please provide your credentials at this AM:'. There are two input fields: 'Username:' with the text 'alice' entered, and 'Password:' with five dots entered. Below these fields is a rounded rectangular button with the text 'Get Authorization'. At the bottom, there is a purple link for 'Debug info:' followed by the text 'Scope: http://localhost:9001/sfs/bob/photos/california.jpg'.

Figure 7.14: User Interface of the requester application that allows the Requesting Party to provide their credentials. These credentials are used for authentication at AM.



The image shows a confirmation dialog box with a light blue border. Inside, there is a white rectangular area containing two lines of text, each preceded by a small square checkbox. The first line reads: 'You must acknowledge to be over 18 years old to be granted access to this item.' The second line reads: 'You must acknowledge to not provide information contained in these documents to any other party.' Below this white area is a rounded rectangular button with the text 'Confirm' in bold black font.

Figure 7.15: User Interface of the requester application that allows the Requesting Party to provide self-asserted claims. These claims are sent to AM for evaluation.

GAE platform to achieve the described functionality.

7.2.8 Implementation

SMARTAM V1 has been implemented in the Java programming language using the popular Spring framework [74; 76]. The User Interface has been implemented using Adobe Flash technology [128]. The API has been implemented using the Apache CXF framework [138] and its JAX-RS component [271]. Security of the AM itself has been provided with Spring Security framework [75], which requires authentication for the implemented User Interface as well as some of the aforementioned API endpoints. Additional security for other API endpoints, as discussed, was achieved using OAuth leeloo [117], which implemented the OAuth 2.0 protocol. SMARTAM V1 can be deployed to a Web container, such as Apache Tomcat [139], and uses a MySQL Community Edition relational database [132] to store information about resources being protected, policies, issued authorisations, and other data.

7.2.9 Limitations

SMARTAM V1 contained a number of limitations that were identified during its development. These shortcomings are discussed in this section. Further analysis of the SMARTAM V1 implementation and its UX is given in Section 7.3 and the shortcomings identified during AM evaluation are discussed in Section 7.3.3.

A list of identified shortcomings is provided below:

1. Lack of support for resource registration via an API;
2. Lack of support for permission registration via an API;
3. Limited support for management of Requesting Parties;
4. Limited support for claims-based policies.
5. Lack of support for federated authentication;
6. Requesters are provisioned with single type tokens only;
7. Support for bearer tokens only;
8. Limited grouping of resources;
9. Lack of support for negative policies;
10. No on-demand sharing of resources;

11. Access to resources via UMA-enabled applications only.

SMARTAM V1 did not support resource registration that could be done by client applications. Therefore, it was the sole responsibility of the user to register resources at the AM using the provided UI. Manual registration of resources is considered unsatisfactory and potentially error-prone. Moreover, host applications did not have information regarding resources that were registered at AM and would refer to the AM for all access requests to resources of a particular user (if such user configured these host applications for UMA).

SMARTAM V1 also did not support permission registration which is defined in more recent revisions of the UMA protocol. This registration prevents the Requester from learning specific permissions that must be assigned to an access token for resources on a Host. Instead, information regarding required permissions to access a resource or a service is shared only between the AM and the Host.

The policy model of SMARTAM V1 allows rules and claims to be included in access control policies. As discussed in Section 7.2.4, a single rule defines a set of Requesting Parties and a set of access rights that these users have. Each Requesting Party includes a set of credentials that have to be provisioned by the user out of band (recall policy examples presented in Section 7.2.6). Such manual management of credentials is inefficient, potentially error-prone, and poses specific constraints.

Firstly, the Authorising User has to create contacts and credentials for their contacts at the AM manually using the provided UI. Moreover, this user has to deliver credentials securely to Requesting Parties. Contacts cannot be shared between users and have to be defined by each user at the AM individually (this is related to the fact that credentials must not be shared).

Secondly, Requesting Parties have to manage credentials for each *{Authorisation Manager, Authorising User}* pair. Moreover, SMARTAM V1 does not implement a password reset function that could be used by Requesting Parties. Therefore, if credentials are exposed, a malicious party can potentially use any application to access resources of the user on a Host. In SMARTAM V1, the end-user is also unable to restrict access to Requesting Parties to use only specific applications when accessing UMA-protected resources. This AM implementation also supports only the username/password flow and the Requester learns the credentials of the Requesting Party at the AM. This requires that only trusted applications are used as Requesters (refer to [202] and [233] for more information on existing threats to clients using credentials to obtain a token from a server).

Policies in SMARTAM V1 can also contain claims. As discussed in Section 7.2.4, a single claim defines a statement that Requesting Parties must assert to the AM. This AM only supports self-asserted claims and the AM only collects user confirmation to those claims. Therefore, it

is assumed that the user tells the truth to the AM before the AM uses the information in the policy evaluation process. For example, the AM can only collect user confirmation that the user is over 18 years old in order to give access to a specific resource.

Moreover, in the presented implementation all claims in the policy have to be provided to the AM. However, it would be beneficial to allow more flexible composition of claims where the Requesting Party could submit only selected claims. For example, the Requesting Party could decide not to provide their confirmation not to print the photos that they are accessing. Instead, the Requesting Party would decide to submit a payment confirmation that would be sufficient to give this user the necessary permissions to access a resource or a service on a Host.

SMARTAM V1 requires both Authorising Users as well as Requesting Parties to sign in to the AM with their credentials. However, federated authentication should be provided to simplify interactions with this system. Currently, the registration process for Authorising Users is manual and requires a developer to create accounts accordingly (recall that Requesting Parties are created by Authorising Users themselves). Moreover, the password reset function is missing. Having a federated authentication mechanism in place would allow to resign from such functionality and to offload authentication to more specialised third party IDPs.

When a Requester applies at AM for access to a resource on a Host, this application is provisioned with a single token (Authorisation Token). The presented AM does not distinguish between tokens that are issued to Requesters for interaction with this AM and then for interaction with host applications. This prevents the Requester to be easily recognisable at the AM which limits the audit functionality that is vital for the end user. Moreover, this AM did not provide any UI for Authorising Users for audit purposes. Therefore, the user was unable to review when a particular policy was created or when a specific resource was accessed.

Moreover, tokens issued for Requesters are bearer-type tokens. This requires Requesters to carefully handle such tokens and not expose them to malicious third parties. Moreover, these tokens are opaque to Hosts. Therefore, the Host has to refer to the AM to check every access request that comes from the Requester. Moreover, the AM issues access control decisions and does not inform the Host about the permissions associated with the token. Therefore, the Host also has limited availability to reuse decisions from the AM for potentially related access requests (e.g. if permissions are assigned for a group of resources, such as all photos in a particular album, then the Host cannot know that and must refer to AM for each access request to every single photo in that album).

In SMARTAM V1, resources are registered as URLs. Therefore, there is limited support for host applications to easily group resources unless the Host can map such URLs internally as necessary. For example, the user can create a resource with a URL of a directory on a Host and

the Host can decide that each file within this directory will be accessed according to that policy.

Because SMARTAM V1 does not support negative policies, then certain use cases and scenarios may need to be implemented using complex positive policies. Once the Requesting Party is provisioned with credentials, it is not possible to easily provide access to all resources apart from specific ones stored on distributed host applications.

Moreover, SMARTAM V1 requires explicit policies to be created in advance before access requests take place. For example, an Authorising User has to first authorise specific Requesting Parties to access their resources and only then these parties can use requester applications. SMARTAM V1 does not support Requesting Parties with applying for access to resources on host applications. Such requests for access would allow the Authorising User to establish policies on demand.

Resources and services can be accessed by Requesting Parties via requester applications that understand the UMA protocol. As such, typical Web browsers cannot act as clients unless these browsers are equipped with some plugins that are able to interact with the Host and AM according to the UMA protocol.

7.3 Evaluation of SMARTAM V1 User Interface

This section presents the evaluation of the SMARTAM V1 User Interface. It discusses the conducted user study, which has been used for evaluation. Based on the gathered feedback, it presents identified shortcomings and derived requirements. These requirements were used in further research on the AM's UX. In particular, these requirements were used for development of a new Authorisation Manager, called SMARTAM V2, which is discussed in Section 7.4. The study in this section is a joint effort with MSc student Iain Carter also published in an MSc thesis in [153], where I was responsible for the design of this study. This MSc thesis presents the complete study and results as well as provides more details than those discussed in this section, including a complete question by question analysis.

7.3.1 Research

In UMA, the AM becomes a central point for managing access to distributed Web resources. Therefore, this system needs to represent a high standard of user-friendliness and its usability becomes a paramount factor.

The difficulty of achieving the usability lies in the distribution of the entire UMA system which comprises of multiple different components, which are host applications, requester applications and the AM itself. The number of parties may cause confusion and a user can get

disoriented easily. Moreover, a user may not easily follow which step should be taken next when managing access control for their resources, taking complexity of the UMA protocol into account. Therefore, the usability factors that the AM should meet are discussed in the next section.

7.3.1.1 Usability Factors

To achieve a satisfactory standard of usability, the Authorisation Manager has to fulfil non-functional usability factors, according to Nielsen [257], such as:

U.1 Learnability - level of ease which lets the user accomplish basic tasks at the first time they encounter the application;

U.2 Efficiency - quality of completing the task, once the user has learned the features of the designed interface;

U.3 Memorability - user's ability to perform particular tasks after a period of time when they did not use the application;

U.4 Errors recovery - design principle letting the user to self correct and get back to the previous stage;

U.5 Satisfaction - quality measured by a user as overall difficulty of use, time it takes to achieve certain tasks and how attractive (both aesthetically and functionally) the service was.

Importantly, Nielsen [256] expresses that usability at a level of 50% is alarming and the one at a level of 66% is average. A research was conducted to verify if the SMARTAM V1 UI fulfils its functional roles within the frames of the aforementioned non-functional usability factors U1-U5. The goal of the research was to test the overall usability score as well. The research method is presented in Section 7.3.1.2.

7.3.1.2 Research Method

A research study was conducted using a controlled experiment followed by a questionnaire. In this study, participants were explained a brief background on the user access control, the outline of the UMA protocol, the UMA AM and what was the intention of the study and the prepared survey. Secondly, the participants were given instructions and were asked to complete a sample task using the implemented SMARTAM V1. This task was based on the scenario, which is presented in Section 7.3.1.3. After completing the user scenario, the participants were requested to fill in the prepared questionnaire. This questionnaire is included in Appendix H.

There were two locations where the study was conducted. Survey participants completed the study either online (*postal-questionnaire*, where respondents answer questions without the aid or presence of the researcher), or completed it in person via an interview (person-administered questionnaire). The use of postal-questionnaire is a low cost method, minimising interviewer bias and is not restricted by participant geolocation [265]. On the other hand, person-administered questionnaires promote a high-response rate and allow interviewer assessment of the user as well as any necessary explanation to the participation [190]. The opportunity for the interviewer to observe the participant while completing the task and questionnaire was also an important factor in choosing the person-administered method. Nielsen states “*Listening to what people say is misleading: you have to watch what they actually do.*” [259]. By observing how participants used the implemented UI of SMARTAM V1, it was possible to collect additional data and gather additional feedback.

Many different aspects of the questionnaire have been specifically designed in order to increase the response rate of the prospective participants. As suggested in [265], the following factors should be taken into account during research: *advance warning*, *publicity*, *incentives* and *confidentiality*. The study presented in this section has addressed each of these four factors. Firstly, an email was sent to candidates giving them advance warning of the study. The study was publicised locally using the aforementioned email distributed to all students in the School of Computing Science at Newcastle University. Furthermore, information regarding the study was placed on the information TV screens throughout Newcastle University and was added to the bulletin board of the UMA WG website at the Kantara Initiative [86]. As an incentive, £10 was offered to anyone who participated in person. Finally, to overcome any potential respondent’s possible apprehensions, each questionnaire featured a small introduction, explaining the purpose of the study, and alerting participants to the fact that all information received from the questionnaire would be anonymous.

The prepared questionnaire contained open and closed questions. Open questions allow freedom of answers, the opportunity to probe and the testing of user ideas or awareness [265], and can lead to answers that were unexpected [190]. The intension of closed questions was to make the task less time consuming to complete it by the participants as well as to analyse, process and compare the gathered data more easily. Each closed question was partnered by an open ended “please explain” question. This allowed the participant to explain their previous choice without being hindered with pre-defined categories, and to fully express freedom and spontaneity in their answers [265].

Questions were ordered so that more personal information came last allowing participants to invest time in the survey before they were asked anything that may potentially scare them away

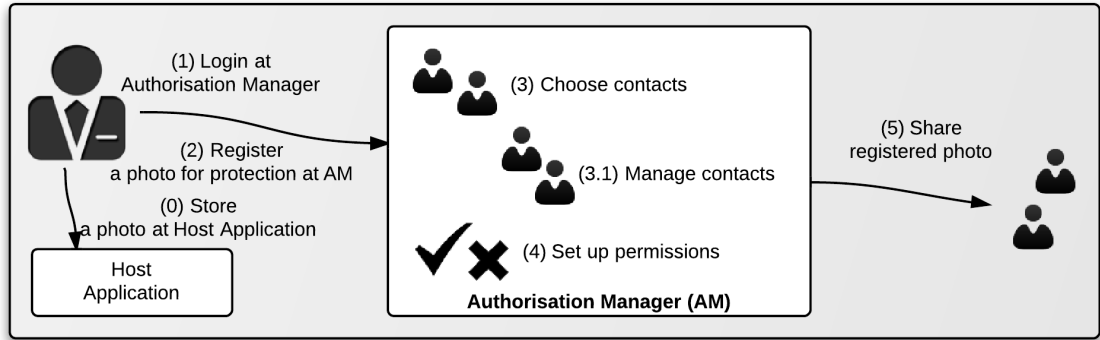


Figure 7.16: User scenario used during research study of the SMARTAM V1 UI.

[185]. Furthermore, Fowler proposes in [190] that the use of coloured paper for questionnaires encourages a higher response rate. Therefore, the online questionnaire was designed with a green background.

7.3.1.3 User Scenario

During the study, users had to complete a well-defined scenario, which is visualised in Figure 7.16. This figure was shown to participants, who completed the scenario using the SMARTAM V1 UI. The task was kept short, but did encompass the use of all areas of the implemented UI in order to receive feedback on all aspects of SMARTAM V1. Importantly, in the study users did only interact with the AM and not with Host or Requester applications. The steps of the scenario are presented below:

1. **S1** Login at Authorisation Manager;
2. **S2** Register a photo for protection at AM;
3. **S3** Choose contacts to share the photo with;
4. **S3.1** Manage contacts if necessary (add new contacts);
5. **S4** Set restrictions to be used in access control policies;
6. **S5** Share the photo by creating an access control policy.

In the presented scenario, two existing applications were used: Facebook [9] and Google Docs [24], which were registered at AM. Importantly, these applications were not integrated with AM (i.e. it was not possible to protect resources stored on those systems) but were presented to users for the purpose of simplicity (it was assumed that users would be more familiar with these applications and not with custom-built UMA-enabled Web applications). The user used

Facebook as a store of their personal information and photos. Moreover, they used Google Docs as an application for managing online documents. As a data owner, the user was asked to share some of their data from Facebook and Google Docs with other Web users, e.g. the user could share their photo. The owner defined who can access what information using SMARTAM V1. Additionally, the owner applied certain sharing restrictions (which were discussed as "claims" in Section 5.7). For instance, they granted access to their data to those users who were over 18 years old.

During the evaluation, the aforementioned scenario was used to collect the data from the study participants. Unfortunately, because of time constraints it was not possible to propose other use cases or scenarios of SMARTAM V1. Such additional use cases would be beneficial and would allow participants to use the implemented system for other tasks than those proposed. This is one of the limitations of the presented AM evaluation.

7.3.1.4 Data Collection

The study was conducted during summer of 2010 and run for a fixed period of 3 weeks. During the study, 34 participants, both men and women, aged between 19 and 50 years old, were interviewed. These 34 responses could be broken down into 30 in person via person-administered method and 4 online via the postal-method. Unfortunately, it was not possible to attract more users to participate in the user study and this is considered as one of the limitations of the presented evaluation.

The responses to the questionnaire had combined both qualitative and quantitative data. The quantitative data collected has been already broken down into specific classifications, as participants were required to select pre-defined responses to each question. However, the qualitative data was subjected to quantification by selecting key trends or exceptions, then such similarities had been combined together. Such groups of trends were coded into numerical values and statistical data had been counted. To quantify data, the Likert Scale [291] was applied. A detailed analysis of coding of the qualitative data is given in [153].

7.3.2 Research Results

This section provides a summary of the participants' answers given in questionnaires and their comments on the implemented UI of SMARTAM V1. In particular, a summary of the feedback given by respondents is given in Section 7.3.2.1. A more detailed analysis is provided in Section 7.3.2.2 and Section 7.3.2.3. A thorough question by question analysis is given in [153]. Findings regarding the UMA concept in general are shown in Section 7.3.2.4.

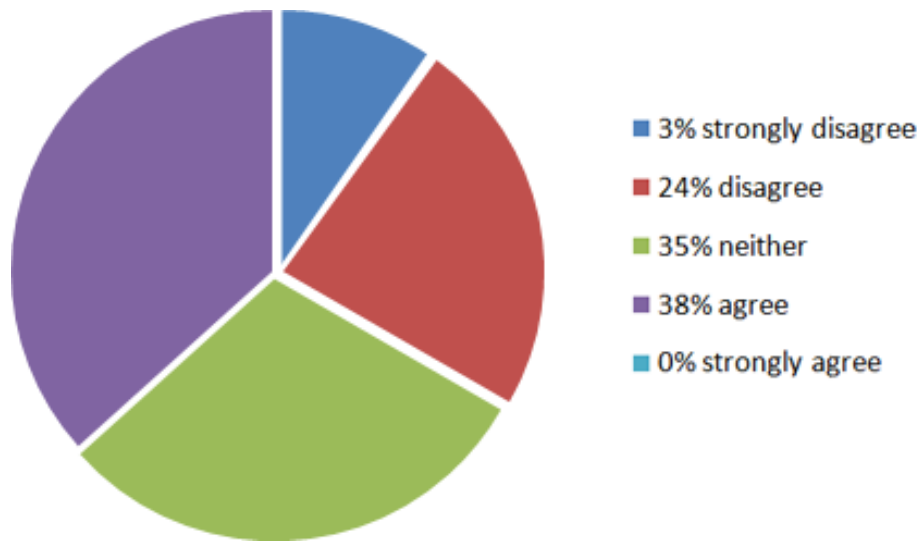


Figure 7.17: Results regarding the general ease of use of the SMARTAM V1.

For the closed-question answers, the Likert Scale [291] was used: *strongly disagree*, *disagree*, *neither agree nor disagree*, *agree* and *strongly agree* were replaced with numbers from 1 to 5 respectively. Open-ended questions were coded so that the opinions contained within them could be analysed. More details are given in [153].

7.3.2.1 Feedback from the respondents

The overall result of the research indicated unsatisfactory level of usability of SMARTAM V1. The general trend was that participants found the AM too complex due to many steps in the process of sharing a resource. They also found it too confusing because of the colour scheme which was difficult to differentiate elements. Furthermore, in such a scheme elements were not sufficiently emphasised. Men tended to find the layout comprehensible; however, they stated that it could have been better. Participants mentioned a confusing order of headlines as a major flaw.

Along with the evaluation of the usability, the research study tried to track users' understanding of the UMA-based approach to protecting and sharing online data. Although respondents understood the idea of protecting resources with a single centralised AM and found it beneficial, they indicated a high level of complexity and difficult use of the proposed UI. Firstly, when the participants were asked if the proposed AM was easy to use, the majority of participants (62%) replied with a score of 3 or less. Only 38% of participants gave 4 in response, agreeing with the statement. Among the respondents there had been no "*strongly agree*" replies. Such a result implied that the UI did not represent a high level of usability [258].

Secondly, the responses to the open-questions suggested that participants had in fact succeeded with their task, but many reported that completion of a task was possible only due to the detailed instructions provided (53% of respondents). This implies that without instructions the task might have been difficult to complete. One participant commented: *"Without initial instructions, I would have been lost"*. The user comments mentioned that the accordion navigation, as implemented in the UI of SMARTAM V1, was difficult to use. One respondent mentioned: *"The tab bars were extremely confusing"*.

7.3.2.2 Interpretation of the results

The interpretation of the obtained results shows poor self-explanation of the implemented UI of SMARTAM V1. From both direct observation of users completing the tasks, and their own comments, an impression might have been drawn that the participants were often unable to locate the resource that they had registered for protection at the AM. However, no hard evidence exists to support this. Any resource added by the user to share with other Web users appears at the bottom of the list in the implemented UI, making it difficult to locate (e.g. one participant commented *"The list of files was difficult to see."*).

Users' difficulty with filling in the form fields proved low learnability. A commonly mentioned issue was that the names of form input fields were not accurate. The reason why participants had difficulties with the text fields was often because they were unsure what to enter due to imprecise corresponding field labels. Terms such as *"name"*, which were required during resource registration, were causing confusion for participants of the study. While observing the participants, one could have seen that they often wanted to enter a person name into these fields (instead of a descriptive and user friendly name of the resource) [153].

The participants found the layout to be not immediately obvious, poorly satisfying the factor of learnability. According to Krug et al., a good usability would require a Web application to be self-explanatory, as discussed in [222]. The most frequent comment of the respondents was that they required a large extent of assistance before they would be able to progress with using the UI. One participant responded: *"The layout is not immediately clear where everything is."* Participants had claimed to depend on the instructions when completing the task of setting a sharing policy for a newly registered data. The minority of 9 participants commented that they would have been unable to fulfill the task without the instructions that they had been given during the study. Explanations made as to what was unclear regarding the layout often included the counter-intuitive order of the headlines which were in different order than the stages of the requested task. Some comments included, for example: *"Headings do not follow order in which they are to be carried out."*

7.3.2.3 Interface inconsistencies

The responses of 79% of participants stated that they noticed inconsistencies while using SMARTAM V1. These inconsistencies were, among others: misleading form field labels, the order of headlines not corresponding to the order of the steps of the sample task, unintuitive drag-and-drop boxes, or the use of the accordion component. One participant commented: *"The list of files is confusing – one tab is already open which meant I didn't even notice."* Participants reported to be particularly confused with the layout of the sharing settings screen. Especially, the drag-and-drop box module caused confusion. The implemented solution of available choices in the right hand column, and a request to drag selected options into the left hand column was found by the participants to be the other main factor leading to confusion. Comments had been made that a more natural solution would be dragging the boxes from left to right rather than from right to left. Especially, in the western world users are used to read from left to right. Human ergonomics in Web application design is essential and not ergonomic design suggests poor usability [248].

In order to improve usability of the proposed UMA system, confusion should be eliminated from the user experience. However one of the comments submitted by the participants was to additionally incorporate an on-screen help to guide users through the process of sharing their online data. Krug et al. [222], however, explains that users are not willing to be confused and all processes ought to be made straightforward. Therefore, if users turned to looking for additional help, it is presumed that the interface was not self-explanatory enough. The new UI and how it addresses this challenge is discussed in further sections of this chapter.

Respondents of the study reported a high level of the error recovery feature. In the opinion of the majority of the participants, they believed that they could easily recover from their mistakes. They had an option to either cancel any command or simply go back if they did a mistake. Nielsen describes error recovery as one of key guidelines to be taken into account when designing any user interface [258]. Because users claimed to recover from mistakes easily, then SMARTAM V1 proved to address the error recovery factor.

The research study showed that the use of colours in the UI was graded equally by the participants. Van Schaik et al. explains in [291] that the use of colour is important to aid the navigation for users. In the study, nearly half of the participants claimed that the interface lacked the necessary use of colours, while the other half accepted the interface as it was. Because many of the respondents believe that the use of colour was poor, this factor was limiting the usability of the implemented UI.

The colour design of the buttons had an instant impact on the users' ability to locate certain objects and process the tasks, as defined by the scenario. Both from observation of participants

and from their own comments, it was possible to reason that the lack of distinction between the background and other structural objects of the UI (e.g. accordion module, etc.) frequently made the participants miss the list of shared resources. Participants reported to be confused as the changes on the list of resources were not distinct enough. They had also declared that more colours would be required while hovering over and selecting items on this list. In the presented UI, buttons and objects change from grey to a lighter shade of grey when being hovered. Participants note that this change is not sufficient to be visually noticeable, and this can lead to confusion.

The majority of the participants found the concept of using the SMARTAM V1 UI comprehensible. Most of them (64%) believed that they knew what stage of the task they were at. While using the AM, they were able to point the stage in the task (as depicted in Figure 7.16).

7.3.2.4 UMA-protocol understanding

Beside the assessment of the SMARTAM V1 UI, the aim of the research study was to measure users' comprehension of the UMA-based access control. Participants understood the approach of the proposed security model with centrally-located Authorisation Manager and they considered it to be advantageous. Participants were satisfied with the functionality of the AM, recognising that the use of it with the ever growing number of Web resources could be helpful. The general concept of the AM itself was frequently highlighted by participants of the research study as one of the key features that they liked the most.

In particular, participants referred to the feature of setting their own restrictions for distributed Web resources. One participant commented: *"The ability to set your own restrictions is useful."* As the AM implements the UMA protocol, the collected results show that participants were complimenting the UMA technology itself and that such technology might not have been difficult to understand. Only 9% of participants found the concept of the Authorisation Manager, and consequently UMA protocol itself, confusing. These participants responded with score 1, indicating that they disagree with the statement that they understood their actions when using the AM. Importantly, no participant responded with score 0 (i.e. *"strongly disagree"*).

7.3.3 Shortcomings and Recommendations

This section maps the obtained results from the study with the defined usability factors U1 - U5, as defined in Section 7.3.1.1. The presented discussion takes into account the Nielsen's Heuristic Evaluation as one of the methods for improving the usability of an interface [258]. In particular, it takes into account the *Consistency and standards* of this method.

A list of identified shortcomings of SMARTAM V1 and its UI are provided with regards to the defined usability factors. Recommended improvements to the AM were derived from these shortcomings. These recommendations were used during implementation of a new AM system, named SMARTAM V2, which is discussed in Section 7.4.

7.3.3.1 Analysis of usability factors

The study of the research results, as presented in Section 7.3.2, illustrates that some aspects of the implemented AM's design support the appropriate usability but only to a certain extent. The intention of the initial design of the UI was to include certain usability factors (as presented in Section 7.3.1.1). The responses received during the study imply that participants found the AM to be consistent (refer to [258] for more details on the importance of consistency within Web domains) and such consistency has been proved to be a key factor providing good usability.

Furthermore, the user interface achieved the goal of proper error recovery (**U4**). The majority of the participants in the study agreed that they were able to easily recover from mistakes. Error recovery is another key usability factor [258].

However, other results of the research study point out that a number of the UI features proved that this UI does not fully support usability. Results found that the design of the UI is counter-intuitive in various places and can be considered illogical. The goal of efficiency (**U2**) was failed to achieve in SMARTAM V1. Weak support of human ergonomics, e.g. right to left drag boxes, is the evidence of poor usability [248]. The use of colour was reported to be limited, whereas colour helps improve both accuracy and the speed of visual recognition when using interfaces [291]. Therefore, wrongly implemented colour was considered as a limitation of the usability of evaluated UI. This suggests lack of user satisfaction (**U5**).

Furthermore, the interface failed to satisfy both learnability (**U1**) and memorability (**U3**) factors. Brief help description was voted the participants' favourite feature. On each page, a help icon offered specific information relevant to common tasks at hand and participants made note of this feature suggesting that it was a useful inclusion to the AM. However, if the design was usable and self-explanatory, participants would not have required any help, thus this indicates poor usability of the user interface and fails to achieve the goal of the aforementioned learnability and memorability. As the users failed to learn the features of the implemented UI, it is also assumed that they would fail to use its features effectively after a break from using it.

The interpretation of the participants' feedback reveals the process of using the Authorisation Manager to be difficult and confusing. Especially the comments provided by 9 participants claimed that they were unable to use the AM unless they were provided with a detailed guidance (such as that included in the task description). In order to prove good usability, any implemented

process should be kept simple [222]. If the process is confusing, it demonstrates poor usability.

As a number of aspects of the AMs design promote appropriate usability, it would be expected that participants' overall impression would be positive. As it was not, the obtained results indicated that the user interface afforded low level of usability.

As discussed in Section 7.3.1.1, Nielsen [258] expresses that usability of 50% is abominable, and 66% is average. It is not suggested that the percentage of people who find the Authorisation Manager easy to use would be the same as the percentage of people who could complete a task in a Web usability test. However, it is possible that these numbers are comparable to some extent. The research results therefore suggest that SMARTAM V1 has poor usability. It is believed that this could be looked into further with specific usability testing, as discussed in [153].

7.3.3.2 Shortcomings

The participants requested that several changes are introduced to the Authorisation Manager. They reported to like the original UI home screen. As a result, the new home screen, as presented in Section 7.4, was based closely on the original design. It was noted that the participants did not like original layout of the headers, and that these headers would need to be rewritten and given more apparent titles. For example, a change from *"People I want to share with"* to *"My contacts"* was requested. These headers were also required to be rearranged in order to logically represent the process of sharing a resource using the AM. The list of shortcomings of SMARTAM V1 and its UI is summarised below:

- SC1** Illogical and counter intuitive design. The order in which the headers were presented on the screen showed poor human ergonomics.
- SC2** Accordion component caused problems for the majority of participants of the study. These participants were unable to locate shared resources and easily define restriction settings for those resources. Accordion was voted as the participants' least favourite feature. Additionally, right to left drag boxes showed poor human ergonomics.
- SC3** Vague names of input fields were confusing for most participants. These fields were used at all stages of the resource sharing task, i.e. while adding new resources, defining settings for those resources and specifying restrictions.
- SC4** Lack of colours to prioritise objects on the page as well as to highlight the selected items or to make links and buttons stand out more clearly.
- SC5** Not enough help was provided during the resource sharing task. The UI was not self-explanatory and such help was necessary.

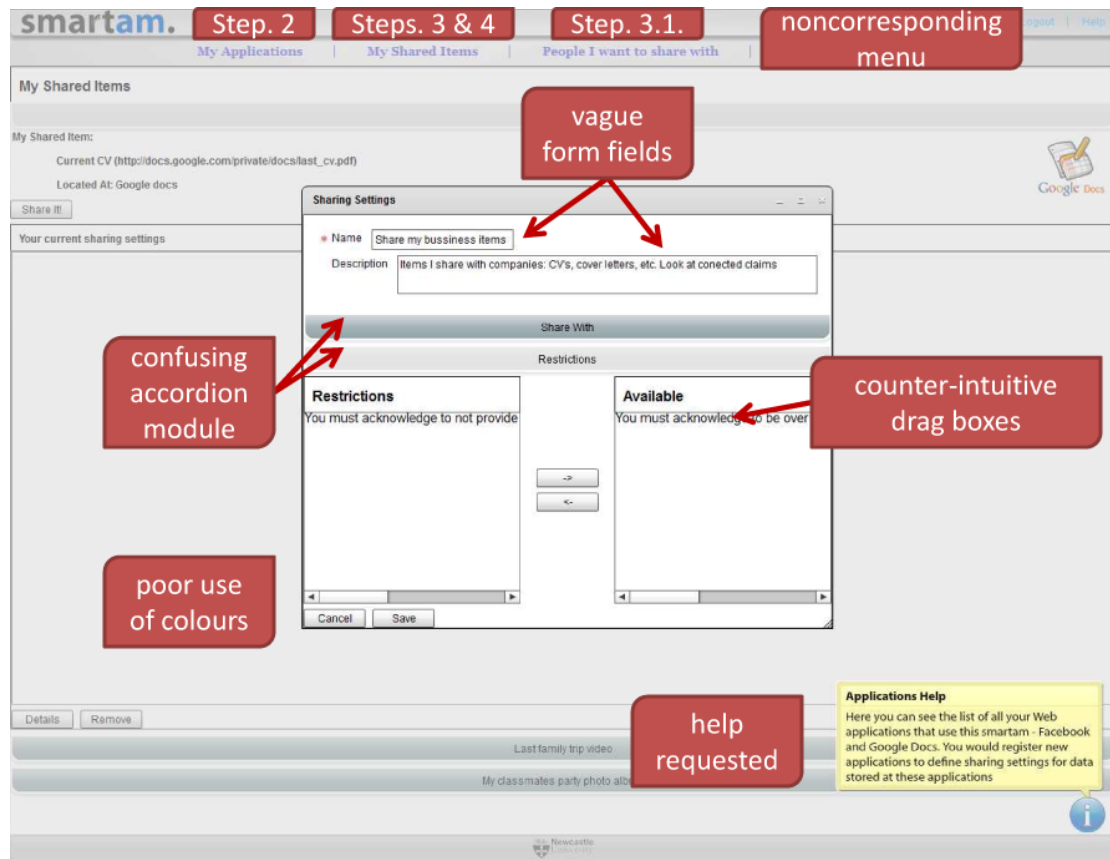


Figure 7.18: Visualisation of shortcomings of the SMARTAM V1 UI.

These aspects of the design have been found by the participants to be of low usability level. These aspects are visualised in Figure 7.18.

7.3.3.3 Recommendations

The UI of SMARTAM V1 had particular aspects that were appreciated by participants (e.g. consistency, easy recovery from mistakes, brief help on various pages). These aspects have been found by participants to be useful and they also coincide with features of a correct, usable design. Furthermore, the UI satisfied all the functional roles required to complete the sample task (as depicted in Figure 7.16). However, the shortcomings of the study results in general indicated a necessity to define new requirements for a User Interface. This was due to the fact that participants reported a poor standard of usability of the UI. These recommendations are presented in the list below:

NR1 Rearrange the order in which the headers will be presented on the screen. The order of the headers should correspond with the order of stages in user scenario and a resource

sharing tasks.

NR2 Accordion component and drag-and-drop components should be exchanged for a simpler visual representation. A simple list or table of items could be a possible substitute.

NR3 Existing labels for input fields should be rewritten to be less confusing. In order to make them more user-friendly, the labels should be accompanied by ordinal numbers, for example: "*1) Choose contacts*", "*2) Choose policies*". Such proposal is likely to cause less confusion among the users and will provide them with clear information regarding the step of the resource sharing process.

NR4 More vivid colour scheme should be chosen to highlight elements of the layout on the page as well as links and buttons that trigger interactions with the user.

NR5 Layout with a higher self-explanatory level should be designed to allow the user, who is unfamiliar with the AM, comprehend what actions they are expected to take. The help feature should be used only as a last alternative.

The aforementioned recommendations are derived from the previously identified shortcomings, i.e. recommendation NR1 is derived from shortcoming SC1, etc. Apart from these recommendations, the navigation in the UI should also be redesigned with the intention of allowing direct access to each of the individual sections of the AM. Furthermore, design should follow closely the guidelines of Nielsen [258] and Krug [222], in particular:

1. Allow the user to easily return to the main page [222] - a home icon is evident on all screens.
2. Adopt an aesthetic, yet minimalist design [222], [258] – text information is kept to minimum with no unnecessary information visible.
3. Use graphical icons [258] – the use of icons has been shown to increase usability as icons help users to recognise different tasks.

To address the identified shortcomings by meeting newly formulated requirements, a new working prototype, called SMARTAM V2, has been developed. This Authorisation Manager is presented in the next section of this chapter. The UI of this new AM is shown in Section 7.4.8.3 based a new and redefined user scenario (Figure 7.25).

7.4 SMART Authorisation Manager V2

SMARTAM V2 is the second, improved implementation of the UMA Authorisation Manager developed during the SMART project. It addresses most of the limitations of SMARTAM V1 as well as shortcomings that were identified during the conducted user study. Furthermore, it provides support for a more recent revision of the UMA protocol. This AM allows the user to compose access control policies and apply them to a set of resources hosted on different Web applications. These applications delegate access control to this AM and are only concerned with enforcing access control decisions. It allows requester applications to obtain authorisations to access protected resources.

SMARTAM V2 was built as a new system and not on top of the first AM implementation. There were three main reasons behind a complete redevelopment effort:

1. Results from the UI evaluation have been obtained;
2. The UMA protocol has changed significantly;
3. A new framework for building Authorisation Managers has been developed.

The list of most significant additions to SMARTAM V2 is given below:

1. Discovery of AM endpoints is based on the newly adopted discovery mechanism;
2. Both host and requester applications can register dynamically;
3. Hosts can register resources using the provided API;
4. Requesters can register permissions using the provided API;
5. AM issues HAT, RAT and RPT tokens to client applications;
6. Policies can be defined for other Web services and other users;
7. AM provides an audit log of data access history;
8. Requester applications can request access to data in the absence of access control policies;
9. Redesigned User Interface provides various ways of showing shared data;
10. Support for third-party asserted claims based on OpenID Connect;
11. Support for federated authentication;
12. Support for management API for mobile applications.

These newly introduced characteristics of SMARTAM V2 are discussed in the next sections.

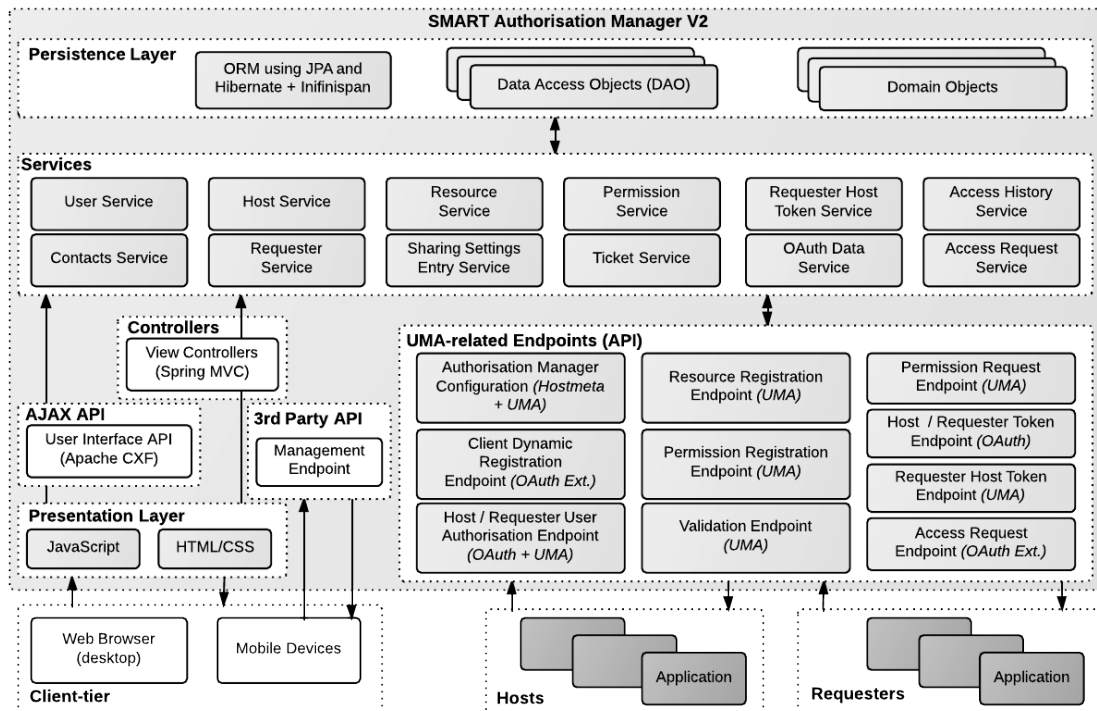


Figure 7.19: Architecture of SMART Authorisation Manager V2.

7.4.1 Architecture

Similarly to the initial prototype, SMARTAM V2 has been implemented as a multi-tier Web application, comprising of the following layers: *persistence layer*, *services layer*, *presentation layer*, and *API layer*. The architecture of the AM is presented in Figure 7.19.

Users of SMARTAM V2 can use the provided UI to manage their registered applications, Web resources, policies for those resources, and contacts. Furthermore, the User Interface has been extended in comparison to the first prototype implementation in order to allow users to review pending access requests to their data (request for access notifications) and to review how their data is being accessed (access history). These two new concepts of the system are discussed in Section 7.4.6 and Section 7.4.7 respectively.

7.4.1.1 Persistence and Services Layer

Data in SMARTAM V2 is processed by different components of the services layer. This layer has been extended in comparison to the first implementation. Firstly, services for managing users (*User Service*) and contacts (*Contacts Service*) have been separated. The *Policy Service* has been substituted with a *Sharing Settings Entry Service* that is responsible for managing individual policy entries. *Resource Service* is now responsible for managing resources as well as

managing policies associated with those resources (see Section 7.4.5). Because SMARTAM V2 supports dynamic resource registration, this service also handles registration of resources via the UMA API.

Permission Service and *Ticket Service* have been developed. Both services provide support for permission registration as defined by the UMA proposal. The *Requester Host Token Service* is responsible for RPT tokens, while the *OAuth Data Service* is a generic service, which supports HAT and RAT tokens for host and requester applications respectively. The *Claims Service* has been removed and its functionality has been substituted with a new endpoint that is discussed further in this chapter.

Additionally, *Access History Service* has been implemented. This service allows users to review access requests to their distributed data. Therefore, users are provided with simple audit log information. The *Access Request Service* allows requester applications to apply, on behalf of Requesting Parties, for access to protected resources in the absence of authorisation policies for those resources.

Services store and retrieve data from a SQL database. SMARTAM V2 uses ORM engine for this purpose. Specific Data Access Objects (DAOs) have been provided, which are used by services to get access to domain objects. For example, the *Resource Service* uses the *Resource DAO* to manage *Resource* domain objects in the persistent store.

7.4.1.2 API Layer

Client applications interact with the Authorisation Manager using HTTP requests sent to the AM's RESTful Web API. This API has been redesigned in comparison to the earlier implementation. Most importantly, this AM supports all endpoints required by UMA *Protection API* and *Authorisation API* as well as common endpoints for Hosts and Requesters.

Firstly, an updated version of the *Authorisation Manager Configuration* endpoint, which now supports JRD-based documents, is provided. Furthermore, *Dynamic Client Registration* endpoint for Hosts and Requesters has been implemented.

Furthermore, an OAuth 2.0 Authorisation Server has been implemented. This server provides the *User Authorisation* and *Token* endpoints that can be used by Hosts and Requesters at SMARTAM V2.

Host applications use *Permission Registration* and *Validation* endpoints, while requesters are provided with *Requester Host Token* and *Permission Request* endpoints. The peculiarities of these endpoints are discussed in Section 7.4.3. SMARTAM V2 also includes an *Access Request* endpoint that allows Requesting Parties to apply for access to protected resources in the absence of authorisation policies.

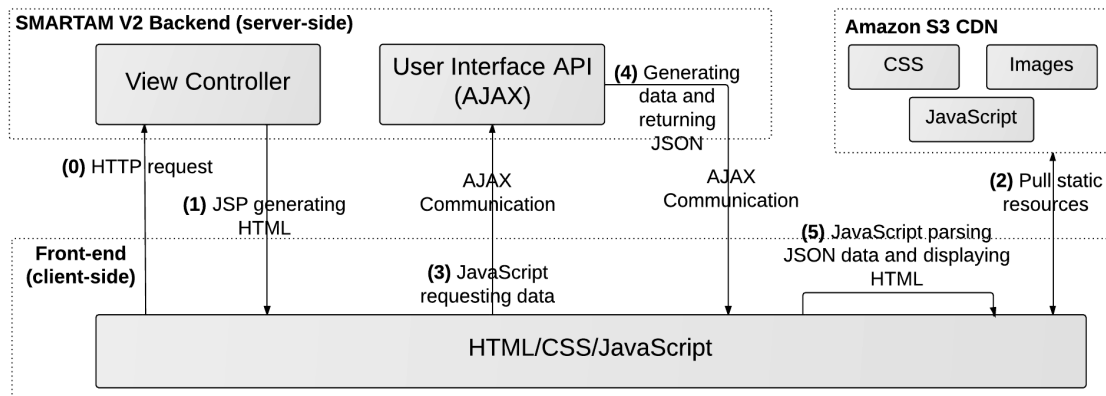


Figure 7.20: Overview of the User Interface architecture of SMARTAM V2.

To allow building mobile applications that could integrate with SMARTAM V2, a *Management* endpoint has been designed. Support for mobile is in its very early stages of development. Therefore, this is discussed as future work in Chapter 8. Importantly, users can still use their mobile devices, such as smartphones or tablets, to access the provided UI of the AM.

7.4.1.3 Presentation Layer

The UI has been implemented using HTML, Javascript and CSS instead of a more complex framework (i.e. Adobe Flex) that was used in SMARTAM V1. The approach of *Responsive Web Design (RWD)* has been adopted and it allowed to implement the AM to be easily accessible from a range number of different devices, including smartphones and tablets. The latter ones, in particular, were becoming more common and the User Interface has been adapted to be accessible on Apple iPad tablets³.

Requests to specific views of the AM (e.g. resources, contacts, etc.) are routed by implemented backend *View Controllers*, which operate on *Models* and return appropriate *Views*. Importantly, SMARTAM V2 has very lightweight controllers only and in most cases these controllers are required to return a view and not provide any sophisticated operations on the model. Instead, the AJAX-based communication is leveraged between the view that is returned to the user's Web browser and the backend API of the AM. For scalability reasons, it was decided to use the popular Amazon S3 service to host static resources. The architecture adopted by the presentation layer is visualised in Figure 7.20.

³A number of presentations showing SMARTAM V2 have been conducted using Apple iPad tablets only.

7.4.2 Security

In order to access the functionality provided by AM, users are required to authenticate first. It has been decided to use a federated authentication protocol for this purpose. SMARTAM V2 allows users to sign in with their Facebook accounts. This functionality uses the OAuth 2.0 derivative protocol, which is implemented by Facebook and discussed in more details in [11]. Therefore, users are not required to register an account manually. Instead, SMARTAM V2 uses the Facebook's Graph API in order to obtain information about a user.

Security of the API is provided with session-based authentication as well as the OAuth 2.0 protocol. API endpoints that are required for user interaction, such as *User Authorisation* endpoints for Hosts and Requesters, require authentication. On the other hand, endpoints that provide functionality to client applications are protected with the OAuth 2.0 protocol. In particular, the following endpoints require an OAuth 2.0 access token to be present in HTTP requests: *Permission Registration*, *Validation*, *Requester Host Token*, and *Permission Request*. The *Management* endpoint for mobile applications requires an API Key to be attached to HTTP requests. API keys are planned to be substituted with OAuth 2.0 in the future. All communication between users and the AM require HTTP protocol with TLS/SSL (although this is deployment specific). The same applies to the communication between client applications and the API of SMARTAM V2.

7.4.3 Support for User-Managed Access

SMARTAM V2 exposes the User-Managed Access functionality through an Application Programming Interface, as shown in Figure 7.19. This software provides this API using a newly developed framework named *UMA/j Authorisation Manager (UMA/j AM)*. UMA/j AM provides implementation of the *Protection API* and the *Authorisation API* which were introduced in Chapter 5. UMA/j AM is itself based on the OAuth 2.0 implementation, which was discussed in Chapter 1. Furthermore, it uses the developed *Discovery* library.

The UMA API provided by SMARTAM V2 comprises of the following parts:

1. Discovery;
2. OAuth 2.0 Client Dynamic Registration;
3. OAuth 2.0 API (*User Authorisation* and *Token* endpoints);
4. Protection API (*Resource Registration*, *Validation*, and *Permission Registration* endpoints);
5. Authorisation API (*Requester Host Token* and *Permission Request* endpoints).

Discovery is available through the *Authorisation Manager Configuration* endpoint, which allows to access a JRD-formatted document that lists endpoints available at AM, supported authorisation types, supported claim types, and other configuration options as discussed in Chapter 5. AM configuration is exposed as a JSON-formatted document (see example in Appendix G).

The *OAuth 2.0 Client Dynamic Registration* allows applications to register themselves as clients of SMARTAM V2 dynamically, using the protocol discussed in [274]. These applications can be provisioned with the necessary OAuth 2.0 credentials, i.e. $\{client_id, client_secret\}$. These credentials are later used during communication with AM. Registration is provided for both host and requester applications. The latter one can also act as anonymous clients as discussed in this section. The discovery and dynamic registration steps are optional and applications can be pre-registered at AM as well.

In order to use the *Protection API* and the *Authorisation API* endpoints by client applications, users have to authorise these applications for the AM. For this purpose, the OAuth 2.0 Authorisation Code Grant [202] has been implemented. In particular, SMARTAM V2 exposes standard OAuth 2.0 endpoints, i.e. *User Authorisation* and *Token* endpoints.

7.4.3.1 Authorisation Tokens

Hosts and Requesters need to obtain access tokens that are authorised for the `uma_host` and `uma_requester` OAuth scopes, respectively. These tokens are named:

1. **Host Access Token (HAT)** (recently renamed to **Protection API Token (PAT)**);
2. **Requester Access Token (RAT)** (recently renamed to **Authorisation API Token (AAT)**).

As discussed in Chapter 5, HAT allows host applications to register resources for protection at AM and to validate access tokens received from Requesters. RAT, on the other hand, allows Requesters to obtain access tokens for protected resources and these tokens are known as **Requester Permissions Tokens (RPT)**. Tokens have been implemented similarly to those used in SMARTAM V1, as discussed in 7.2.3.1.

Access tokens are accompanied with additional information, namely the expiration time given as a Unix timestamp as well as corresponding refresh tokens. Importantly, access tokens are valid for 3600 seconds (1 hour) and have to be refreshed by client applications. Refresh tokens, on the other hand, are long-lived and have to be revoked by users manually. The implemented client frameworks, which were discussed in Chapter 6, can automatically detect validity of access tokens based on their expiration time, and can refresh these tokens without any intervention from the application developer.

7.4.3.2 Support for Host Applications

Host applications have to be registered as OAuth 2.0 clients in order to interact with the *Protection API* provided by SMARTAM V2. Hosts have to be authorised for access to those endpoints by Authorising Users. This authorisation is obtained using the OAuth 2.0 Web Server flow, which uses the provided *User Authorisation* and *Token* endpoints at SMARTAM V2 and allows the Hosts to obtain HATs.

The Protection API is composed of the *Resource Registration*, *Permission Registration*, and *Validation* endpoints. The first one supports host applications with registering resources for protection. This is a major difference in comparison to the first AM implementation that required manual resource registration using the UI. This endpoint conforms to the one discussed in Chapter 5 but additionally allows host applications to obtain a URL of the access control policy, that is associated with a resource after registration. This is discussed separately in Section 7.4.3.3.

Hosts register permissions required by requester applications using the *Permission Registration* endpoint of SMARTAM V2. Implemented *Validation* endpoint supports host applications with evaluating access requests to protected resources. This endpoint conforms to the one discussed in Section 5.4.5. The Host sends the RPT, along with its own HAT, in order to receive permissions associated with this RPT. Therefore, the Host can make a meaningful decision whether access to a resource should be granted or not. Policy evaluation is discussed in Section 7.4.6.

7.4.3.3 Resource Registration

SMARTAM V2 implements the resource registration protocol provided by the User-Managed Access proposal. This differs from SMARTAM V1, which required users to register resources manually. Users can interact with the AM directly from their host applications. In particular, they can select which resources should be protected and host applications can register such resources at runtime. This functionality has been provided using the UMA/j AM framework. Chapter 5 gives detailed explanation of the resource registration protocol.

When a user creates a resource at a Web application then this resource is either private or public. Such setting depends solely on the design choices and implementation of such Web application. If access control is delegated to the implemented AM then the default policy for registered resources is set. Currently, the resource is automatically associated with an empty policy and advanced settings must be set manually by a user. Automatic policy associations are planned to be introduced in further versions of the presented AM implementation. Policy templates which support such functionality are given in Section 7.4.5.

The basic proposal of resource registration has been extended to support access control policies, which are currently left outside of the UMA scope. This extension allows the AM to register a resource for protection and then inform the Host about the policy in place. This functionality is necessary in order to support users with their interactions with existing Web applications. In particular, introducing this feature allows to address the issues, which are identified in the scenario in Section 1.1.2 from Chapter 1.

When a resource is first registered, the AM creates a link to the policy for this resource and returns this link back to the host application. The host application can associate such link with the resource itself on its side. Chapter 6 discussed how the Puma framework uses this policy link to provide a simplified user experience for managing access to Web resources.

7.4.3.4 Support for Requester Applications

Requester applications are not required to be registered as OAuth 2.0 clients and these applications can act as anonymous clients to the AM. These applications can interact with the provided *User Authorisation* and *Token* endpoints providing their obtained credentials or *{anonymous, anonymous}*. When these applications interact with the AM without registration, no refresh token is issued for the corresponding RAT and these applications have to go through the authorisation every time the RAT expires. Support for anonymous clients is planned to be removed.

Requesting parties are required to authorise their requester applications for the *Authorisation API* provided by SMARTAM V2 before they can access UMA-protected resources. This API, composed of the *Requester Host Token* and *Permission Request* endpoints, allows these applications to obtain the necessary RPTs and take part in the authorisation phase. Requesters first obtain empty RPTs using the implemented *Requester Host Token* endpoint and only later associate necessary permissions with this RPT using the *Permission Request* endpoint and newly introduced *Access Request* endpoint. Importantly, as defined by UMA, Requesting Parties may need to provide claims before the RPT can be associated with necessary permissions.

It is important to note that the scope of access for the RPT is specified by the Authorising User in the policy associated with a resource (see Section 7.4.5). Depending on the policy, the requester application that interacts with the *Permission Request* endpoint can either obtain an upgraded RPT (if the policy allows for it) or can be provided with the *claims requested* document, which lists the necessary claims that must be submitted by the Requesting Party.

SMARTAM V2 implementation supports only one type of claims that a Requester can encounter: `redirect_required`. When such claim is received by the Requester, then the Requesting Party is redirected to the location contained in the claim. Such redirect is provided to support the following cases:

1. Requesting Parties that are also owners of protected resources can authorise their requester applications to access these resources *ad hoc*;
2. Requesting Parties that are not owners of resources can provide additional OpenID Connect claims to authorise their requester applications. This holds for access control policies that include entries which require claims;
3. Requesting Parties that are not owners of resources can request access to protected resources at AM.

In the aforementioned use cases, after the redirect is complete, the Requesting Party is redirected back by the AM to the requester application's callback URL. Then Requester can proceed depending on the AM's response. This callback URL is discussed using an example in Section 6.3.4.4. The response returned by the AM to the Requester using the post-claims HTTP redirect includes such information as the state, which the Requester can use to pull information on the original request to access to data, and either of the following parameters issued by the AM:

1. **x-oauth_access_granted** - informs a Requester whether access was granted or not.
2. **x-oauth_access_req** - informs a Requester whether access to a resource was requested at an AM or not.
3. **x-oidc_claims_submitted** - informs a Requester if a Requesting Party has successfully submitted necessary claims to an AM, allowing a Requester to continue access to a protected resource.

If the Requester finds the **x-oauth_access_granted** parameter in the HTTP request, then it checks if this parameter has the **true** value. The presence of this value means that the user has authorised the Requester to access their protected resources on the Host. This is the *Person-to-Self* sharing scenario, where the Requesting Party is the same as the Authorising User.

On the other hand, if the Requester finds the **x-oauth_access_req** parameter set to **true** in the HTTP request, then it knows that a request for access has been submitted to AM. This situation occurs when the access control policy has not yet been setup for the Requesting Party, who seeks access to a protected resource. This is the *Person-to-Person* or *Person-to-Service* sharing scenario. The Requester can present a custom UI to the Requesting Party and can inform the Requesting Party that request for access has been submitted successfully.

Furthermore, support for OpenID Connect claims in SMARTAM V2 has been implemented. Therefore, the Requester must be able to respond to the **x-oidc_claims_submitted** parameter

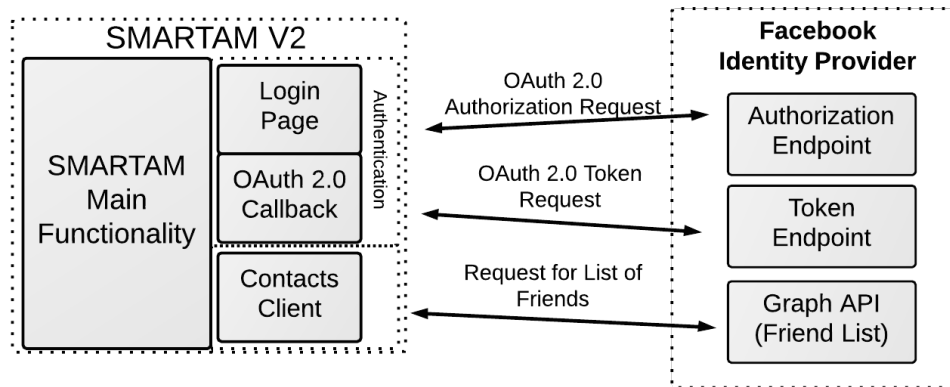


Figure 7.21: SMARTAM V2 integrated with Facebook.

set to `true` in the HTTP request. Such parameter occurs in *Person-to-Person* sharing scenarios, where the Requesting Party successfully submitted necessary claims to the AM using the OpenID Connect protocol.

In cases where either `x-oauth_access_granted` or `x-oidc_claims_submitted` parameters are returned with their value set to `true`, then the RPT is considered to bear new permissions and the Requester should try to access the resource again.

Importantly, the aforementioned parameters, `x-oauth_access_granted`, `x-oauth_access_req`, and `x-oidc_claims_submitted`, have been introduced in the implementations presented in this thesis. These parameters are not part of the UMA protocol specification. Examples of how these parameters are used in the discussed software are provided in Section 6.3.4.4.

7.4.4 Federated Authentication

SMARTAM V2 supports federated authentication. It allows users to sign in with their Facebook accounts using the OAuth 2.0 derivative protocol implemented by this IDP. Furthermore, users can use Facebook to import their contacts using the integrated Graph API [10]. Each imported contact is created as an unregistered user and is linked with the original user through a notion of a *contact*. Users are then able to create new access control policies using these contacts. The integration between SMARTAM V2 and Facebook is visualised in Figure 7.21.

Moreover, the AM implementation allows users to define access control policies with manually registered contacts. In such cases, these contacts may need to authenticate to the AM using the OpenID Connect protocol. For example, if a policy includes an entry that would allow access to a person who owns the *bob@gmail.com* email address, then this person has to sign in with their OIDC-compliant Google Identity Provider.

7.4.5 Policy Model

Resource sets registered at SMARTAM V2 are always associated with an access control policy. Such policy was initially called *Sharing Settings*⁴ and was composed of multiple *Sharing Settings Entries* (which are equivalents of access control entries). The *Sharing Settings* and *Resource* database tables have been eventually merged. Each registered resource at the AM now contains the list of policy entries that define how this resource is shared.

When the resource is first created then the list of entries is empty by default. The AM provides a UI to create new entries that define how a resource is shared. Each entry can specify an application, which can act as a Requester to access the data. Also, each entry can specify a user and application pair, which can act as a Requesting Party and Requester pair that can access the data. Entries can belong to a single policy only and cannot be shared across multiple policies. This limitation of the AM implementation is discussed further in this section.

SMARTAM V2 supports the following policy types (which are in fact policy entries):

1. **Myself** - contains the list of requester applications and their permissions for a resource;
2. **Others** - contains the list of Requesting Parties, associated with requester applications, and their permissions for a resource.

The *myself* type policy is provided for *Person-to-Self* sharing scenarios, which are similar to OAuth where resource is shared between services on behalf of the same user. The UI for this policy type is presented in Figure 7.36. A detailed explanation of possible sharing scenarios in UMA is given Section 5.3. For example, a user may authorise their online photo editing tool to access their photos stored in their online gallery service. Another example is where the user authorises an online job application system to access their verified "Transcript of Records" document stored on their Personal Data Store application. Implementation of the latter scenario is given in Section 6.3 and screenshots of this flow are given in Appendix I.

The *others* type policy allows users to share data with other users on the Web. Importantly, these users have to use applications that can access UMA-protected resources. As such, typical Web browsers cannot act as clients unless these browsers are equipped with some plugins that are able to interact with the Host and AM according to the UMA protocol.

User interfaces for the *others* policy type are shown in Figures 7.37, 7.38, and 7.39. This policy type supports *Person-to-Person* and *Person-to-Service* sharing scenarios where a resource is shared between services and the Requesting Party is different from the Authorising User. For example, a user can share their photo album with other users, irrespectively of the application

⁴In this section, the terms "access control policy", "policy" and "sharing settings" are used interchangeably.

that they use (e.g. such album will be accessible by specific users over the API through a number of different applications). Another example is where a user shares their verified personal information with a potential employer, who uses a specific online job application system to access such data. Other examples have also been discussed in this thesis (see Chapter 1 and Chapter 5).

Importantly, myself- and others-type policies support restricting access to specific actions on resources. Such actions are resource set specific and are dynamically registered by Hosts at the AM. These actions are represented as string values and can be arbitrarily complex (e.g. `delete` and `perform_analysis`). These actions were discussed in Section 5.4.2.

Resources that are protected by AM are considered to have a *custom* access control policy, which is defined by the entries in such policy. However, the user can still make registered resources publicly available without disassociating them from AM. The implemented UI allows users to mark a resource as *public*. On the other hand, if the resource should be made *private*, the UI supports such function as well. Importantly, marking a resource as public or private does not modify existing entries in the access control policy but results in the *Validation* endpoint returning either *"positive"* or *"negative"* responses to host applications. Furthermore, requester applications that want to associate necessary permissions with their RPTs will either have these permissions associated automatically, or will be automatically rejected. By allowing users to create entries in their access control policies as well as to quickly mark resources as publicly available or private using the AM, the user has flexibility in how their online resources are managed. In particular, users can use the AM to make resources temporarily private, adjust policies as necessary, and only then share these resources with other users and services on the Web.

The user can also mark resources public or private at the Host, which would allow the Host to resign from consulting the AM about the access rights of these specific resources. In such case, the Host would not have to make the extra effort of contacting the AM as long as it knows the correct access rights of the resource. The discussed model does not prevent this situation where users define access rights locally at host applications and this can be more efficient in terms of communication overhead. However, the user would lose all benefits of the UMA approach, including the ability to have a holistic view of the applied security controls over their distributed data.

SMARTAM V2 implementation does not allow for policy reuse across different resources. In particular, it is not yet possible to share a single policy. Such functionality existed in SMARTAM V1 (recall Section 7.2.4). Despite this limitation, the presented policy model can be considered as very flexible and being able to meet numerous user needs for complex online transactions.

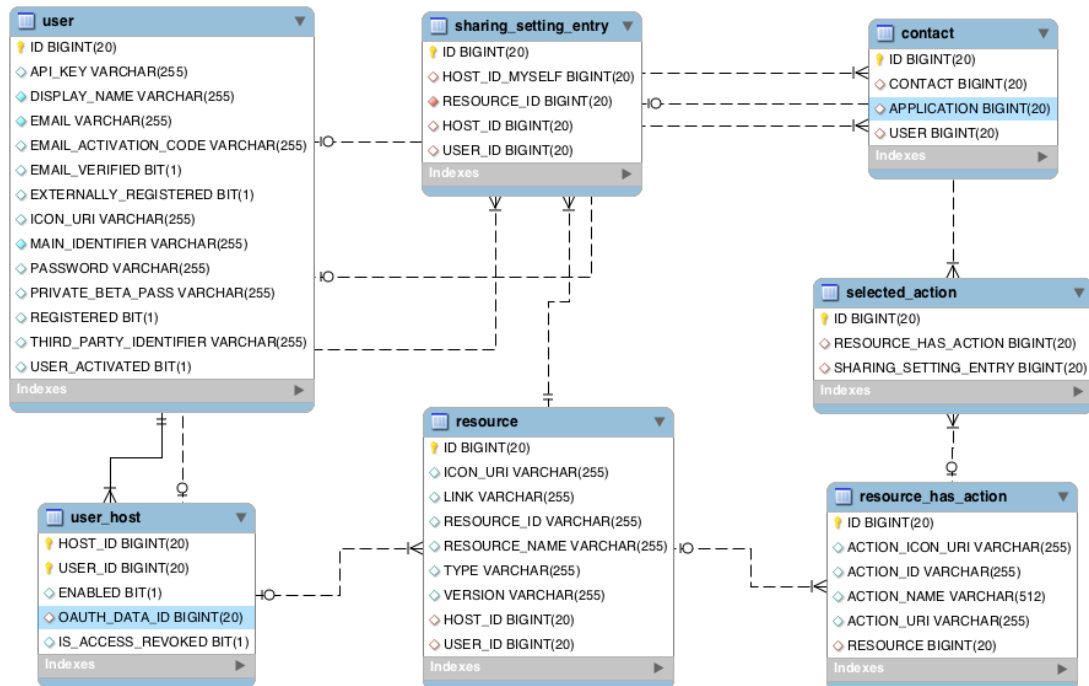


Figure 7.22: Entity Relationship Diagram of access control policies in SMARTAM V2.

Shared policies are planned to be introduced. Such policies would allow users to create reusable *policy templates*. Importantly, the presented policy model in AM has been designed to support such templates.

7.4.5.1 Database Implementation of Policies

In order to understand the policy model available in SMARTAM V2, this section presents the database implementation. It shows how different entities of the DB model are related to provide the described functionality (see Figure 7.22).

user - list of users in the Authorisation Manager. *one-to-many* relationship with the **user_host** table, that links users and their registered applications (either Hosts or Requesters);

contact - links two different **user** entries together;

resource - *one-to-many* relationship with the **sharing_setting_entry** table, describes the resource being protected at AM;

ID - internally used identifier of the resource;

HOST_ID - ID of the Host application, which registered this resource for protection;

USER_ID - ID of the Authorising User, which is the owner of the registered resource;

RESOURCE_ID - externally visible resource ID as registered by the Host;

TYPE - specifies the security of the resource - **PRIVATE**, **PUBLIC**, or **CUSTOM**;

sharing_setting_entry - stores policy entry for a single *{application, user}* pair;

HOST_ID_MYSELF - ID of the Requester for which this policy is defined. Set only if the policy is defined for the Authorising User ("myself-type policy"), otherwise it is a policy for others (refer to **HOST_ID** and **USER_ID**). It has to be equal to the **USER_ID** column in the **resources** table that it references;

HOST_ID - ID of the Requester for which this policy is defined, which will be used by Requesting Parties other than the Authorising User ("others-type policy");

USER_ID - ID of the Requesting Party, for which this policy is defined ("others type policy"). This identifier references the **contact** table;

RESOURCE_ID - internally used ID of the resource; points to the **resource** table;

resource_has_action - list of available actions associated with a resource;

selected_action - list of actions associated with the **sharing_setting_entry** (actions from *myself* or *others* policy).

In others-type policies, entries use the **contact** table to specify Requesting Parties. Importantly, this table has a *recursive relationship* with the **user** table. Therefore, it links two user entities together. This is a typical parent-child relationship, with an Authorising User being the parent and the contact user being the child.

Importantly, the entries in the **resource** table have the **TYPE** column, which is used to define how a policy should be treated at the AM. If the resource type is set to **PUBLIC**, then no entries are checked during permission requests and validation processes. Therefore, requester applications can have their RPTs associated with permissions without any form of authorisation. Additionally, host applications are always provisioned with a *"positive"* validation response from AM. However, if the resource type is set to **PRIVATE** then the resource itself is not accessible (irrespective of whether Requesters were issued any RPTs for this resource, since the RPT validation will simply fail). The **CUSTOM** type means that individual entries for a resource should be checked.

Furthermore, the AM implementation has defined the following identifiers in the **sharing_setting_entry** table as *special cases*: **HOST_ID** = 10 (meaning *"Any Requester Application"*) and **USER_ID** = 10 (*"Any Requesting Party"*). Both identifiers were implemented to support the following cases:

1. *Resource is public* - the resource is accessible using the UMA protocol by any Requesting Party that uses any Requester application ($HOST_ID = 10$ and $USER_ID = 10$). This is the equivalent of marking the resource as public but it distinguishes the actions for which a resource is made public;
2. *Person-to-Person sharing* - the resource can be accessed by a specific Requesting Party using any Requester application ($HOST_ID = 10$ and $USER_ID \neq 10$);
3. *Person-to-Service sharing* - the resource can be accessed by any Requesting Party using a specific Requester application ($HOST_ID \neq 10$ and $USER_ID = 10$).

7.4.6 Policy Evaluation

SMARTAM V2 evaluates policies before RPTs are issued to requester applications or associated with necessary permissions. Furthermore, the validation of these tokens is considered as part of the policy evaluation process, too.

7.4.6.1 Issuing empty RPTs to Requesters

The Requester has to be in the possession of an RPT in order to access protected resources at a Host. Therefore, as defined by the UMA protocol, it uses one of the *Authorisation API* endpoints to obtain such a token for a specific Host.

SMARTAM V2 provides a *Requester Host Token* endpoint, which issues empty RPTs. This differs from an early UMA proposal, where RPTs were issued by the *Permission Request* endpoint (see Chapter 5 the UMA protocol specification in [297]). Importantly, the RPT is initially associated with the $\{Requesting\ Party, Requester, Host\}$ tuple.

7.4.6.2 Registering Permissions

Requesters use the newly issued RPTs for access requests to protected resource. Since these RPTs are not yet associated with any permissions, then such requests will simply fail. However, for every request that is accompanied with an RPT, the Host registers the necessary permission that would be required for such access to succeed.

SMARTAM V2 provides host applications with the *Permission Registration*. Registration requires providing the identifier of the resource being accessed as well as the set of actions, which need to be authorised for the Requester. This information is used in further phases of policy evaluation at the AM. An example permission registration request is given in Listing 8. Such registration results in the permission ticket being returned to the Host and eventually to the Requester.

Requesters use the permission ticket, along with their RPT, at the *Permission Request* endpoint. The purpose of this endpoint is to allow the Requester take part in the authorisation phase. An example request is shown in Listing 23.

```
1  POST /api/uma/permissions_grant/ticket/3a055317... HTTP/1.1
2  Authorization: Bearer 7e6e5d10e725482c34a5ac59a4b37b2b
3  Accept: application/uma-claims-required+json
4  Content-Type: application/json
5
6  { "rpt": "908ba7ad3fea56a15fcaa73248ac9ba7" }
```

Listing 23: HTTP request of Requester applying for permissions to be assigned to an RPT.

SMARTAM V2 uses information from the request to find the registered permission and to retrieve the resource and its associated access control policy entries. This functionality is provided by the *Resource Service* and *Sharing Settings Entry Service*. Both services evaluate access requests against defined policy entries and can result in one of the following responses returned to the Requester:

1. HTTP 401 **Unauthorized** - the policy specifies that the resource is private;
2. HTTP 201 **Created** - response containing a newly created RPT, with associated permissions for a resource if either: (1) the policy specifies that the resource is public, (2) the policy contains an entry that matches the registered permission;
3. HTTP 200 **OK** - response containing the *claims requested* document.

Furthermore, in case there was a problem with the request, the AM responds with the HTTP 400 **Bad Request** response. In particular, such error is returned if there was a problem with the permission ticket (e.g. this ticket might have been already used or it cannot be found).

7.4.6.3 Obtaining Authorisation

The AM supports only one type of claims in the *claims requested* document: **redirect_required** (see example in Listing 24). When such claim is returned to the Requester, then the Requesting Party should be redirected to the location contained in the claim value. Importantly, SMARTAM V2 distinguishes a case where a policy contains an entry with a manually added Requesting Party. Such manually added party contains an email address. This email address can be used as an attribute in the policy evaluation process. In such cases, the redirect URL in the **claim_value** field contains the **claims_ext_openid** parameter set to **true**. The decision to use the

email as the attribute was dictated by the adoption of the OIDC protocol in SMARTAM V2 implementation. This protocol is used to gather claims of the Requesting Party and email is among the standard OIDC claims that is required to be returned by OIDC IDPs [58].

```
1  { "requested_claims": [  
2      {  
3          "claim_type": "redirect_required",  
4          "claim_name": "Redirect Required",  
5          "claim_value": "https://www.smartam.org/request_access/  
6          entry?x-request-access=true&  
7          scope=35f4b49e2bbde56631e777dfc8789e8b  
8          &response_type=code&x-type=popup  
9          &redirect_uri=http%3A%2F%2Fpds.com%2Fa%2Fuma%2Fredirect  
10         &claims_ext_openid=true&client_id=23663654000000"  
11     } ]  
12 }
```

Listing 24: Claims requested document issued by AM.

The redirect URL in the `claim_value` field points the Requesting Party to the *Access Request* endpoint. This endpoint is not a standard UMA endpoint and has been introduced in the discussed AM implementation. Importantly, this endpoint supports three different cases that differ based on the party that is redirected to this endpoint. In particular, the following parties are distinguished at the *Access Request* endpoint:

1. Authorising User acting as Requesting Party;
2. Requesting Party added manually by Authorising User;
3. Any Requesting Party.

7.4.6.3.1 Authorising User acting as Requesting Party. When the Authorising User equals the Requesting Party, then the *Access Request* endpoint displays a page for the user, where such user can authorise their requester application for a specific access type. Such page resembles a standard OAuth 2.0 authorisation page, which provides information regarding the client application, the resource as well as the set of actions which will be authorised (please note a difference between this page and a standard OAuth 2.0 page, which only informs the user about the client application and a set of OAuth scopes). Example of such page is presented in Figure 7.23. Importantly, the user has to be signed in to the AM.

When the user authorises their Requester to access a protected resource on their behalf, a corresponding entry in the access control policy for this resource is created. With such approach,

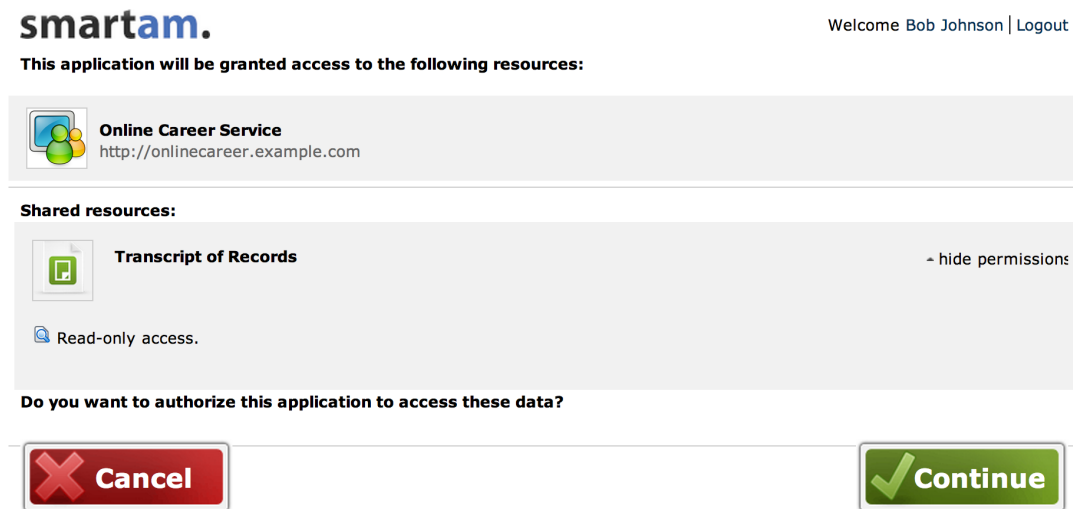


Figure 7.23: Authorisation page that allows the Requesting Party to authorise a requester application to access a protected resource stored on a Host.

the user can establish relationships between their online applications on an *ad hoc* basis. Policies do not have to be specified in advance. Furthermore, the AM informs the Requester about granted access using the `x-oauth_access_granted` parameter in the post-claims redirect. This approach differs from the classic OAuth, because relationships between different applications are stored centrally and not on distributed OAuth Authorisation Servers.

7.4.6.3.2 Requesting Party added manually by Authorising User. The policy may contain an entry that specifies a Requesting Party, which was added by an Authorising User manually. In such case, a Requesting Party is presented with a choice to sign in to the AM using the OIDC protocol. SMARTAM V2 currently supports a very basic attribute model by requiring Requesting Parties to provide verified email addresses. Furthermore, this AM supports only the Google IDP, which was one of the existing OIDC implementations available at the time when this feature was implemented.

With OpenID Connect claims, the Requesting Party signs in to the OIDC Provider and authorises the AM to access their profile information. Because SMARTAM V2 supports email attributes, authorisation for both `openid` and `email` scopes is required [57]. When the AM is able to verify that the email address obtained from the *UserInfo* endpoint of the OIDC provider matches the email specified in the policy, it can then upgrade the RPT that belongs to the requester application. Requester is informed about granted access using the `x-oidc_claims_submitted` parameter, which is returned using the front-channel communication (i.e. via the User-Agent).

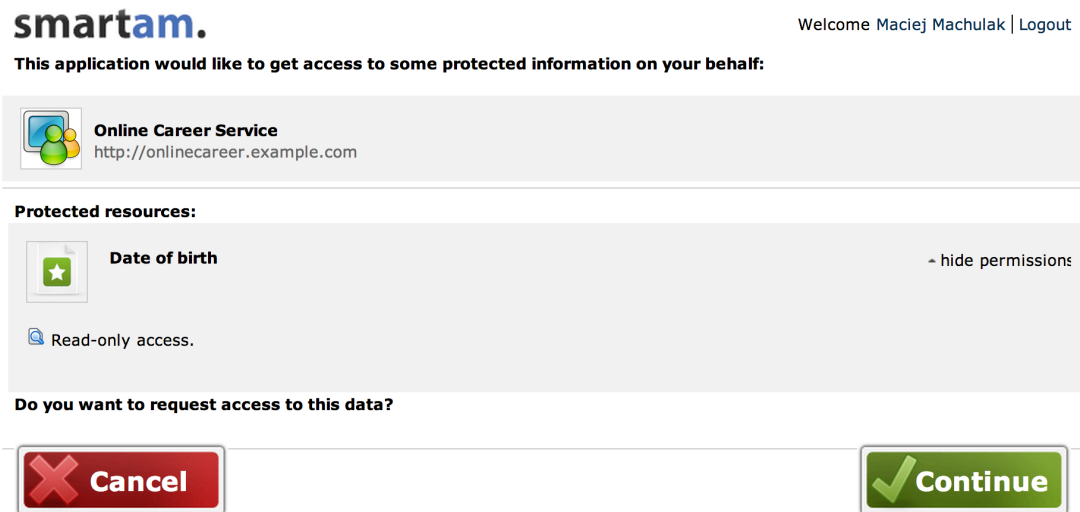


Figure 7.24: Request for access page that allows the Requesting Party to request access to a protected resource stored on a Host.

7.4.6.3.3 Any Requesting Party. When the *Access Request* endpoint is accessed by any Requesting Parties, the AM presents a *Request for Access* page (see Figure 7.24). This page is not a part of UMA but has been introduced in the presented AM to support use cases, as discussed in Chapter 1. This page resembles the OAuth authorisation page. However, it does not authorise any access but merely asks the Requesting Party if they wish to request access to a protected resource. This request is stored in a persistent store on AM, which may also send an email to the user who owns the data informing them about a new pending access request. Importantly, no data is shared without the user's explicit consent. After successfully submitting this request for access, the AM redirects the Requesting Party, using front-channel communication, to the requester application and includes the `x-oauth_access_req` parameter set to `true`. This parameter informs the Requester that access to a resource has been requested. Therefore, the Requester can act accordingly, e.g. periodically check if access was approved.

7.4.6.4 Validating Tokens from Requesters

Policy evaluation is also performed when the host validates the received RPT from the requester application. Requests for validation are sent to the AM's *Validation* endpoint. This endpoint uses a custom `AMValidationProvider` that implements a simple logic, which returns either a *public authorisation*, *none authorisation*, or a *custom authorisation*.

Custom authorisation depends on the RPT presented by the Host to AM and the resource for which access is requested. Firstly, the `AMValidationProvider` checks if the Authorising User and the Requesting Party are the same and if an entry already exists for this user. Depending

on the existence of the entry, either a *custom* or a *none authorisation* is returned to the client application.

On the other hand, if the Requesting Party is different from the Authorising User, then the provider uses the implementation of the `SharingSettingsEntryService` interface to check policy entries using either:

1. `findByAccessTokenResourceIdByClientId();`
2. `findForAnyUserAnyHost().`

The first method searches for the RPT token data, which was originally issued to the Requester. It additionally matches such data with existing policy entries for the resource stored on the Host. This results in a single entry being returned by this method. The `findForAnyUserAnyHost()` method is a fallback mechanism that is used to check if the Authorising User has defined a resource to be accessible by any $\{Requesting\ Party, Requester\}$ pair. If no entries are found, a *none authorisation* is returned to the Host.

7.4.7 Access History

In Chapter 1, the following requirement for a user-managed access control solution was provided: "*A consolidated view of the applied security controls as well as audit information should be provided to support users with managing access to the ever-growing number of resources on the Web.*". To meet this requirement, a new feature has been introduced to the presented AM named "*History*". With this new functionality, users can quickly review how their distributed resources are accessed by Requesting Parties and Requesters. This feature is provided using a separate UI view. This view presents a list of entries representing individual access requests to protected resources.

Each entry in the list is created during the RPT validation process. A single entry specifies a request, which resulted in the *public authorisation* or *custom authorisation*, as discussed in Section 7.4.6. Such entry contains the following information: Requesting Party, Requester, resource being accessed, Host that stores this resource, time of access.

SMARTAM V2 issues tokens that are opaque to Hosts. Therefore, RPT validation has to be performed every time a host application receives an RPT. This results in such validation being logged at the AM and it is visible to the user through the provided UI. Currently, only successful access requests are logged but failed requests are planned to be logged as well.

The discussed "*History*" view currently supports access requests to UMA-protected resources only. This view is presented in Section 7.4.8.3 in Figure 7.45. However, this view is planned to be extended to support other events of the AM. In particular, it is planned to include policy

specification related events, such as creation or deletion of a new entry in a policy for a particular resource or resource registration and de-registration events.

7.4.8 User Interface Description

The UI has been redesigned based on the feedback gathered during the initial user study, as presented in Section 7.3. Furthermore, it included new features, based on additional feedback gathered during the research and development of the UMA protocol and implementation of this protocol. This UI is believed to satisfy the usability factors, which were presented in Section 7.3.1.1, more successfully. In particular, the new design is more intuitive in comparison to the previous UI. The layout of the elements is more logical and wizard-type solutions have been introduced.

Furthermore, a single screen layout is used in the system. In the original design in SMARTAM V1, users were required to switch between different screens (recall the *"My Shared Items"* and *"People I want to share with"* screens) when they wanted to set up a single policy for a resource. In the new design, a user has a single screen where they can view a resource that is being protected by AM and is presented with a simple choice of myself- or others-type policies. Myself-type policy allows to display a list of requester applications and their permissions. Others-type policy allows the user to view the list of Requesting Parties, their associated requester applications, and their permissions.

The interface also provides users with access to history of access requests that were issued by Requesters to protected resources. Furthermore, it supports users with creating policies at runtime for their Requesters as well as for those Requesting Parties that apply for such access.

7.4.8.1 Scenario

This section presents the implemented UI based on a new and redefined generic user scenario, which includes new features of the UMA protocol. These features did not exist during the original user study that was discussed in Section 7.3. In particular, these features, visible to the end-user, include dynamic resource registration that was not implemented in SMARTAM V1. Federated authentication has been also integrated with SMARTAM V2.

The new scenario is depicted in Figure 7.25. It includes the following steps:

1. **NS1** Login at Host and AM;
2. **NS2** Register a document for protection at AM using host application;
3. **NS3** Choose contacts to share resources with;

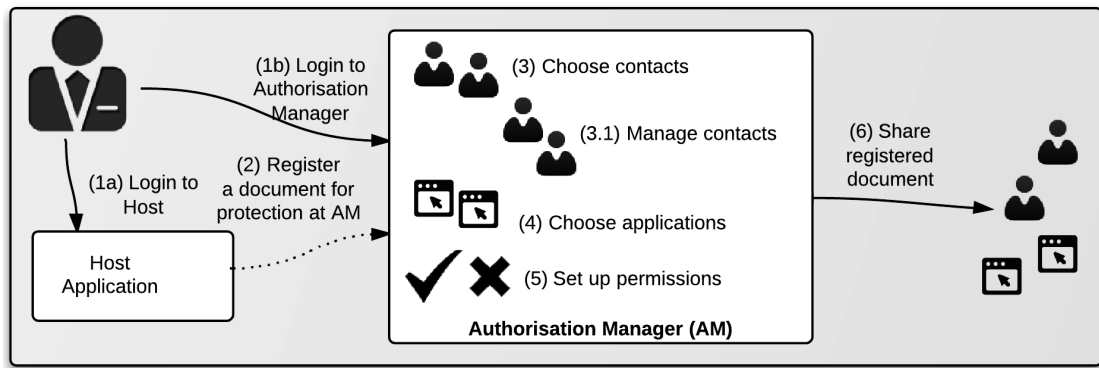


Figure 7.25: User scenario used for discussion of the SMARTAM V2 User Interface.

4. **NS3.1** Manage contacts if necessary (add new contacts);
5. **NS4** Choose applications to be used by contacts for accessing the protected resource;
6. **NS5** Setup permissions for contacts and applications;
7. **NS6** Share the document.

Registering resources and available actions for those resources is now moved to the host application where the data is stored. This step is now a part of the UMA protocol that was discussed in Section 5.4.2. For the purpose of the scenario, it was decided to use the developed applications, namely *PDS* application as well as the *CareerMonster* application (see Appendix D).

7.4.8.1.1 Login at Host and AM. In the new scenario, the user completes the first step **NS1** by signing in to the *PDS* and to the Authorisation Manager. Typically, the user would login separately at the PDS and separately at the AM. Then, following the UMA protocol, the user would need to establish a trust relationship between the PDS and the AM, which would result in the PDS being provisioned with the HAT/PAT token that this application can use at AM. However, a new User Experience simplification has been tested, which would allow the user to sign in to a Host (e.g. PDS) and additionally achieve the following: (1) the user gets signed in to the AM without any additional authentication or authorisation processes, and (2) the Host obtains the HAT/PAT token to use the AM's Protection API. This simplification was tested using a third party identity federation proxy and it is not discussed further in this thesis.

7.4.8.1.2 Register a document for protection at AM using host application. At the PDS, the user is able to protect their personal information and the "Transcript of Records" data, among other resources, with the implemented AM. After selecting the sharing option, available

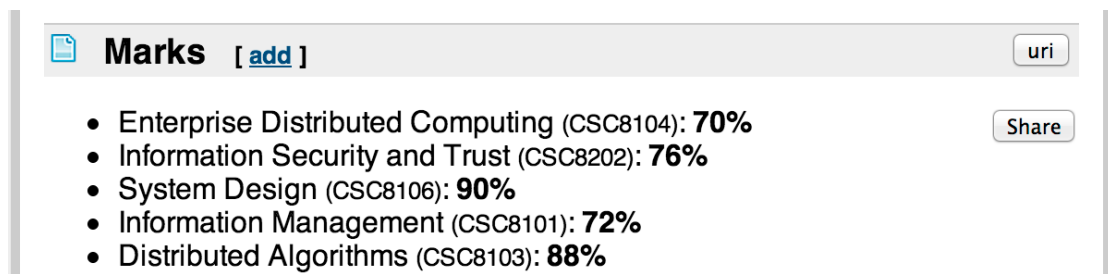


Figure 7.26: Resource and a Share button rendered for this resource on a Host.

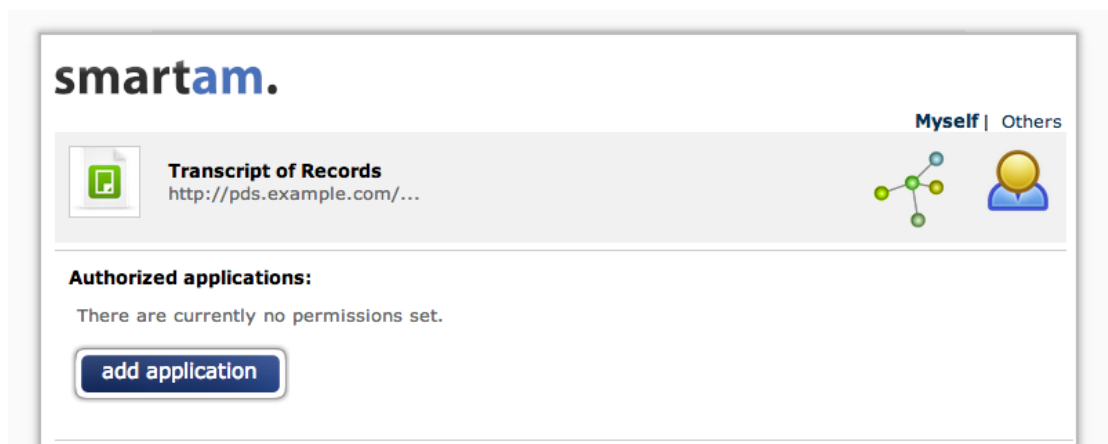


Figure 7.27: View presenting myself-type policy provided by SMARTAM V2 and accessed directly from a host application.

through the "Share" button, the user is redirected from the Host to the AM (see Figure 7.26). This way, the second step **NS2** in user scenario is completed automatically (in comparison to manual resource registration, as presented in Section 7.3.1.3). In particular, the resource is registered at AM and the PDS application obtains information about the policy location for this resource. This policy is displayed for the user directly from the host application.

The redirect of the user to the policy location was initially done in a pop-up, which was displayed on top of the original UI of the Host. Therefore, the user had the impression that this policy is a part of the host application. However, the user could still verify that it is their AM that was being used for policy specification. For example, the URL of the AM being accessed was displayed in the address bar of the pop-up window. The user could also verify that the access was done through HTTPS (see Section 5.4.1). In further versions, the redirect of the user to the policy is provided within an `iframe` and not within a pop-up. Examples of policies loaded within an `iframe` are shown in Figure 7.27 and Figure 7.28.

At the policy view, the user is presented with an icon for the resource that they are protecting with AM, and myself- and others-type policies for this resource. Importantly, each resource is

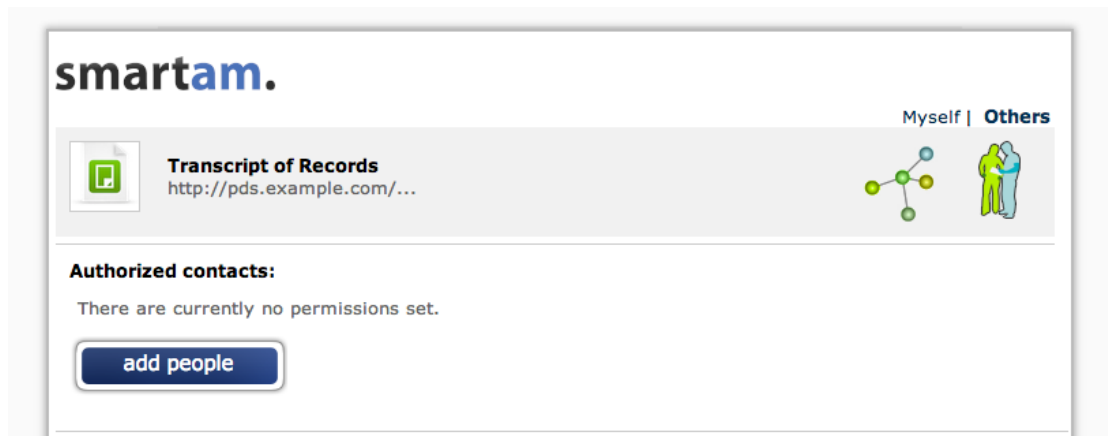


Figure 7.28: View presenting others-type policy provided by SMARTAM V2 and accessed directly from a host application.

associated with a single policy only and such policy can contain multiple different entries. Each entry is marked as either myself- or others-type and the view simply filters entries based on the type selected by the user. By default, when the policy is accessed from the Host then the myself-type policy is displayed (see Figure 7.27).

When the resource is first registered, there are no entries in the access control policy for this resource. In the discussed scenario, the user shares the data using the others-type policy, which is more complex and allows to discuss more details of the implemented AM. In the others-type policy, the user can add Requesting Parties and Requesters. In the myself-type policy, only Requesters can be added.

As discussed in Section 7.4.5, policies can also specify resources as public and private through the *"Make Public"* and *"Make Private"* options. Currently, both options are only available using the main interface of the AM and are not displayed when the policy is accessed directly from a host application.

7.4.8.1.3 Choose contacts to share the document with. In order to add new entries in the others-type policy, the user is required to click on the *"add people"* button, which constitutes step **NS3** of the discussed scenario. This results in an overlay with a form being displayed to the user. In order to make the form more straight forward, it has been designed to be more visual and not to resemble a usual Web input form. Each step of creating a new policy entry is initially hidden and is displayed only if the previous step has been completed. Furthermore, the field labels have been rewritten in comparison to SMARTAM V1 (see Figure 7.30). By taking such measures, the form resembles a wizard form with each step of the policy specification process clearly indicated to the user.

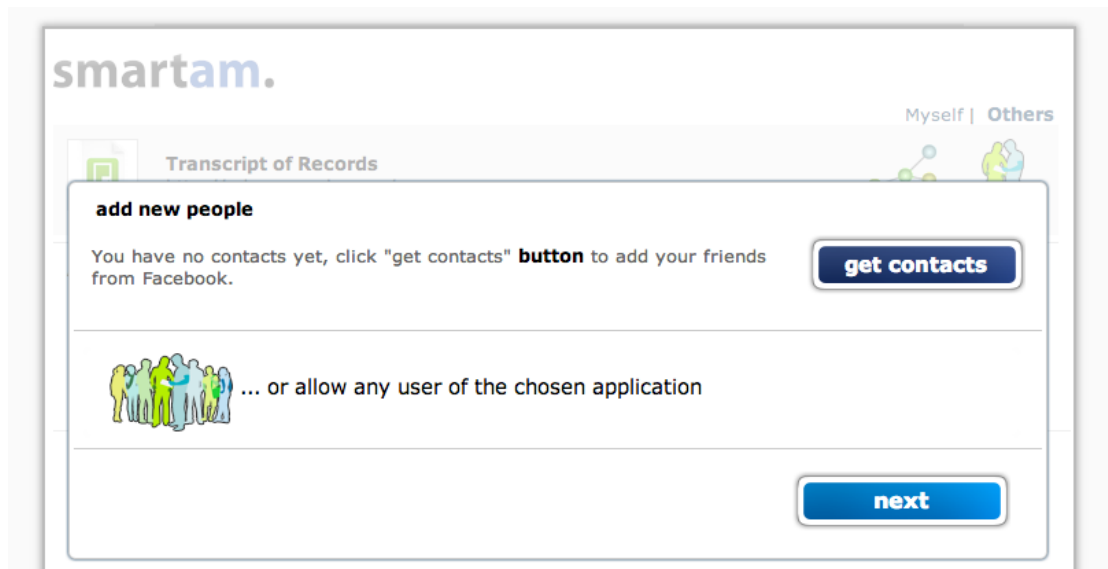


Figure 7.29: Access control policy view with missing contacts.

SMARTAM V2 has been integrated with the Graph API provided by Facebook. Therefore, it supports importing contacts from Facebook and associating them with the user at AM. The relationships between the Authorising User and potential Requesting Parties were discussed in Section 7.4.5.1. The user can import contacts from Facebook into the AM by clicking on the *Get Contacts* button (see Figure 7.29).

7.4.8.1.4 Choose applications to be used by contacts for accessing the document. After selecting the contacts, the user is provided with the next view of the policy specification step. The user can select applications which can be used by previously selected contacts to access the resource. This is step **NS4** of the scenario. Furthermore, the user can also select that Requesting Parties will be able to access resources using any application. Such access can be also restricted by specific actions selected in the policy entry. Importantly, the user has to select applications first before it is possible to select any actions for these applications.

7.4.8.1.5 Setup permissions for contacts and applications. When applications and actions are selected, the user can add the new entry to the policy (Figure 7.31). This step **NS5** ends the process of defining an access control policy entry for a resource. Once the user adds this new entry to the policy, then the overlay is closed and the user lands on the view of their resource at the PDS. On selecting the *"Share"* option again, the user is shown the screen and can verify that the policy has been created.

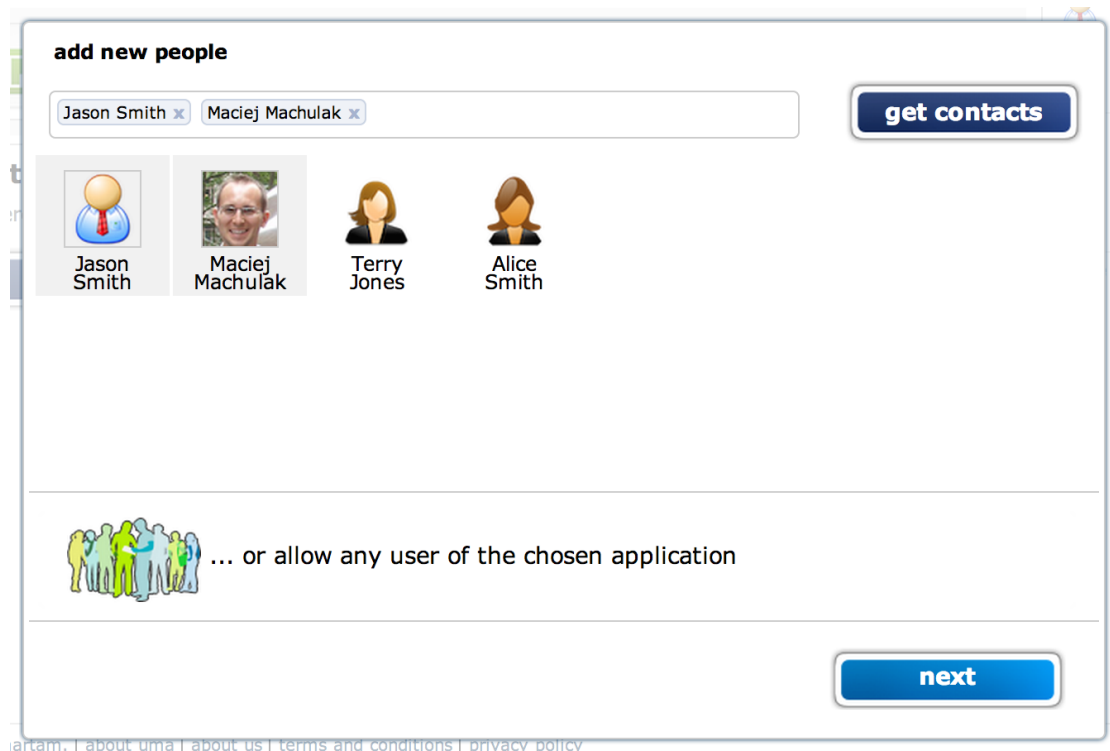


Figure 7.30: View for selecting contacts to be used in a policy.

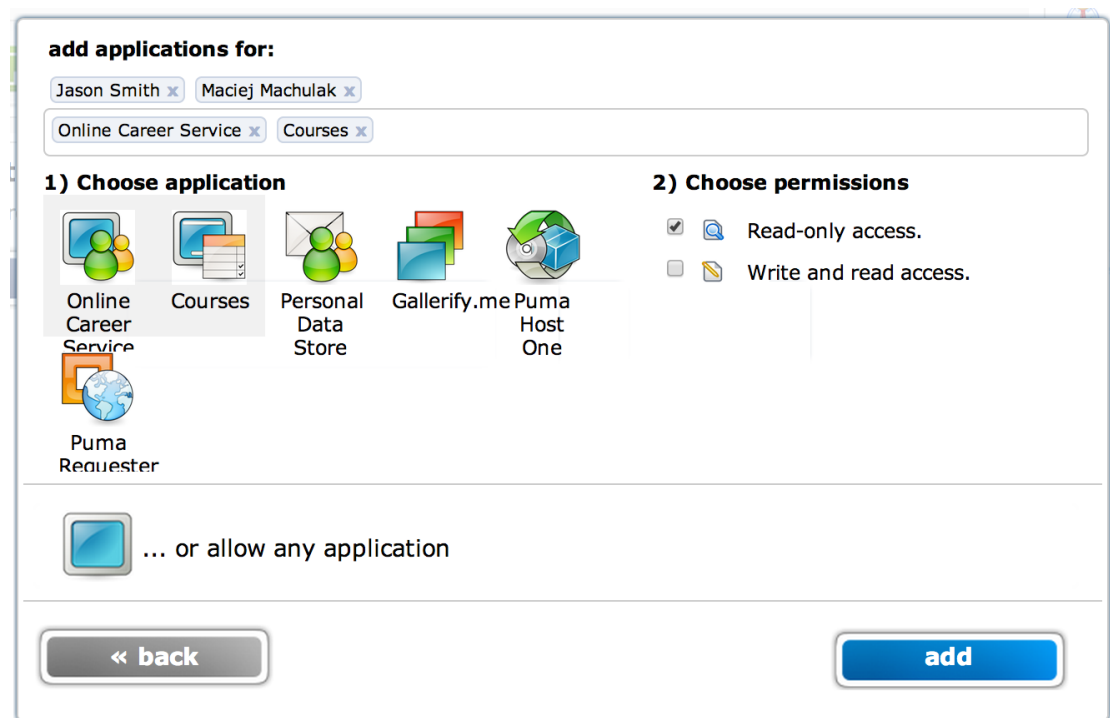


Figure 7.31: View for selecting applications and actions for these applications to be used in a policy.

smartam.

Dear Bob

A new resource has been shared with you by: Alice

Resource name: dogs (Gallerify)

You can access the resource using: Gallerify, SmartFetch

Resource link: <https://www.gallerify.me/g/alice/albums/dogs>

**Thank you,
The SMARTAM Team**

Figure 7.32: Example email that could be sent to a Requesting Party notifying them about a newly shared resource.



Figure 7.33: Information sent to the Facebook wall of a Requesting Party notifying them about a newly shared resource.

7.4.8.1.6 Share the document. When a new entry is created in a policy for a resource, then the Authorisation Manager could send an email to the Requesting Party with whom the new resource has been shared. Such email could contain information regarding the application that can be used to access the resource as well as the resource location. Unfortunately, it has not been possible to implement sending of such emails yet and the Authorising User has to inform the Requesting Party about a newly shared resource or a service out of band (e.g. by manually sending an email). This step **NS6** ends the newly defined scenario for the AM's UI.

Emails could be sent only to those contacts that have their email address specified. For example, contacts imported from Facebook do not have email addresses. Therefore, a different approach of notifying users about newly shared resources would be necessary (Figure 7.33).

Users who are provided with information about newly shared resources have to be cautious when accessing applications or links that are referred to in the received notifications. As discussed

in Section 5.4.1.4.1, there is a possibility of phishing attacks. For example, an attacker can send out spam emails containing URLs to his own (attacking) Web sites rather than genuine ones. Such attacks may trick users to reveal confidential information (e.g. username and password) to malicious third parties. Therefore, it is important for the user to understand and recognise the legitimacy of the application they are accessing. For example, the user needs to check that the URL in the address bar in the browser matches the URL of their legitimate client application or their legitimate Authorisation Manager. However, this thesis does not focus on the phishing problem and makes an assumption that the user is conscious about the application they are interacting with.

7.4.8.2 Main User Interface

The AM can be also accessed in a standard way and not by initiating the flow at the host application. As such, it provides Authorising Users with a Web-based administration panel, which UMA calls the *"Privacy Dashboard"*. The user can sign in to the AM with their Facebook account, as previously discussed.

At the main view, the AM presents a list of protected resources stored on registered host applications. This view is called *"Data"* (Figure 7.34). Users can manage either myself or others type policies for each registered resource (Figure 7.35). Importantly, this view is the same as the one accessible directly from the host application. In particular, it allows users to access myself and others type policies. However, it also introduces the options to make the resource private or public (such setting takes precedence over entries in the policy). The user can revoke access rights by deleting individual entries in the policy.

As discussed in the previous section, the myself type policy shows a list of requester applications that are authorised by the user to access protected resources on behalf of this user. Each requester application is authorised for specific actions which can be executed on these resources (Figure 7.36). The others type policy shows a list of Requesting Parties with whom a resource has been shared, including the applications and authorised actions.

Instead of a confusing accordion module and drag-and-drop boxes, nested lists are displayed for both myself and others type policies. On the myself type policy, at the first level, there is a list of authorised requester applications. The user can select each application and is provided with the list of authorised actions for this application (Figure 7.36). On the others type policy, at the first level there is a list of Requesting Parties, i.e. contacts (Figure 7.37). Next to each contact there is a list of requester applications displayed at the second level (Figure 7.38). At the third level, there is a list of authorised actions (Figure 7.39).

Apart from specifying policies for their online resources using the Web interface provided by

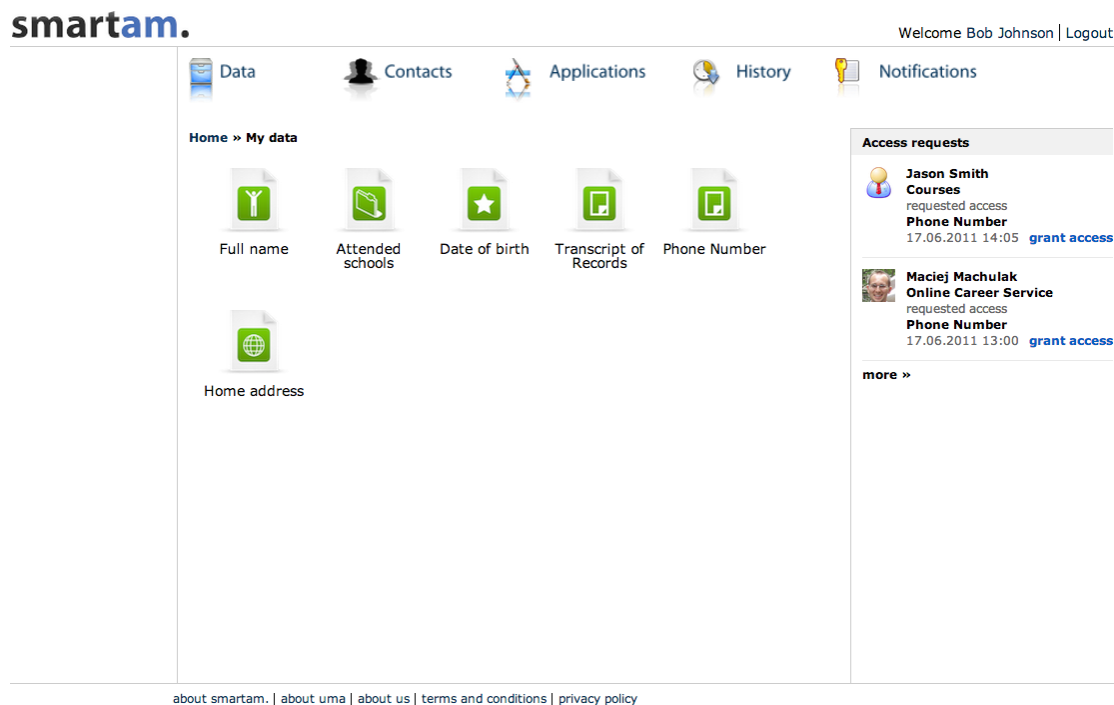


Figure 7.34: Main view of SMARTAM V2 presenting resources registered for protection.

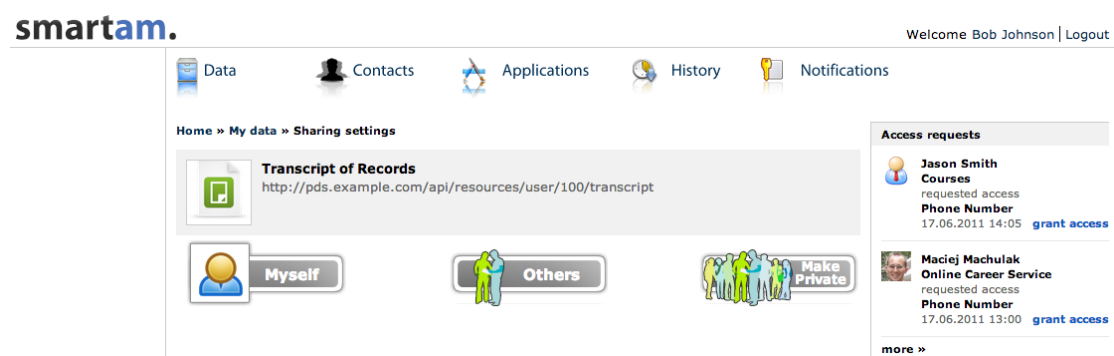





Figure 7.35: View presenting the resource registered for protection on SMARTAM V2.


Home » My data » Sharing settings for Myself

Myself | Others

**Transcript of Records**
http://pds.example.com/...




Authorized applications:

**Online Career Service**

hide permissions ✕ remove

☒ Read-only access.
☐ Write and read access.

save »

**Courses**


see permissions ✕ remove



add application

Figure 7.36: View presenting a specified myself type access control policy.


Home » My data » Sharing settings for Others

Myself | **Others**


**Transcript of Records**
http://pds.example.com/...



Authorized contacts:

**Jason Smith**

see permissions ✕ remove

**Maciej Machulak**


see permissions ✕ remove



add people

Figure 7.37: View presenting a specified others type access control policy.


Home » My data » Sharing settings for Others

Myself | **Others**



**Transcript of Records**
http://pds.example.com/...




Authorized contacts:

**Jason Smith**- hide permissions ✖ remove

Jason Smith`s applications:


Online Career Service Courses

add application


**Maciej Machulak**- see permissions ✖ remove



add people

Figure 7.38: View presenting a specified others type access control policy with application details.


Home » My data » Sharing settings for Others

Myself | Others



 **Transcript of Records**
http://pds.example.com/...



Authorized contacts:

 **Jason Smith** - hide permissions ✖ remove


Jason Smith`s applications:

 
Online Career Service Courses

Online Career Service`s permissions:

☒  Read-only access.
☐  Write and read access.

[add application](#) [save »](#)

 **Maciej Machulak** - see permissions ✖ remove

[add people](#)

Figure 7.39: View presenting a specified others type access control policy with application and permission details.

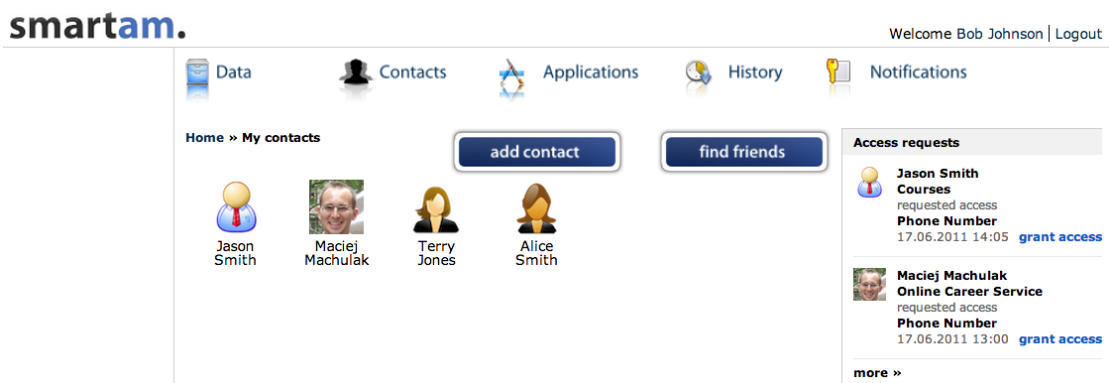


Figure 7.40: View showing the list of contacts.

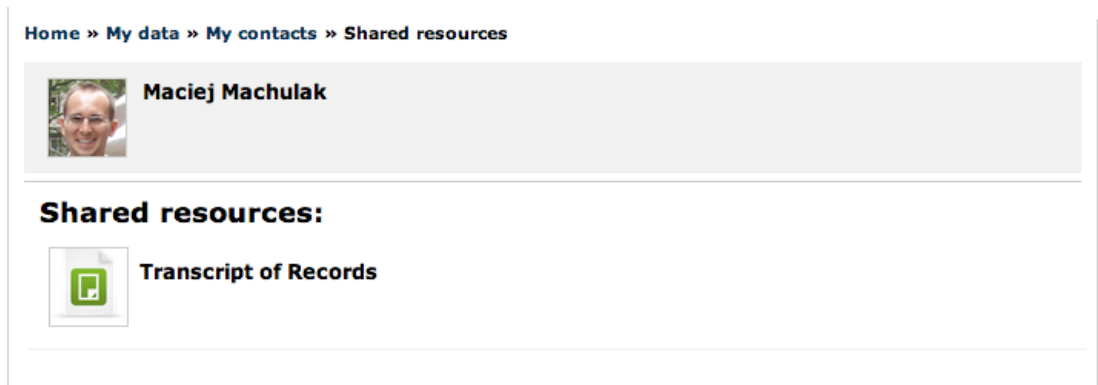


Figure 7.41: View presenting resources shared with a Requesting Party.

AM, the user can also manage their contacts, which can be used in access control policies. The UI with the list of contacts is presented in Figure 7.40. The contact list, when accessed directly from the AM side, has more features in comparison to access done from host applications. On selecting a person from contact lists, the contact details are displayed. The user can also view which resources can be accessed by this particular contact (Figure 7.41).

By default, users can import their contacts from Facebook. SMARTAM V2 also supports manual contact registration. Manually registered contacts can be later used for OpenID Connect-based claims in access control policies. For example, sharing with *bob@gmail.com* requires the Requesting Party to simply authenticate with an identity associated with such verified email address. This functionality has been implemented using the OpenID Connect protocol. The user can add contacts by providing a name and email address (Figure 7.42).

A new view has been also implemented to allow users to see applications available at SMARTAM V2. The *"Applications"* view provides a list of all applications, including Hosts or Requesters (Figure 7.43). When the user selects an application then they can review which resources can be accessed by this application on behalf of the user (Figure 7.44).

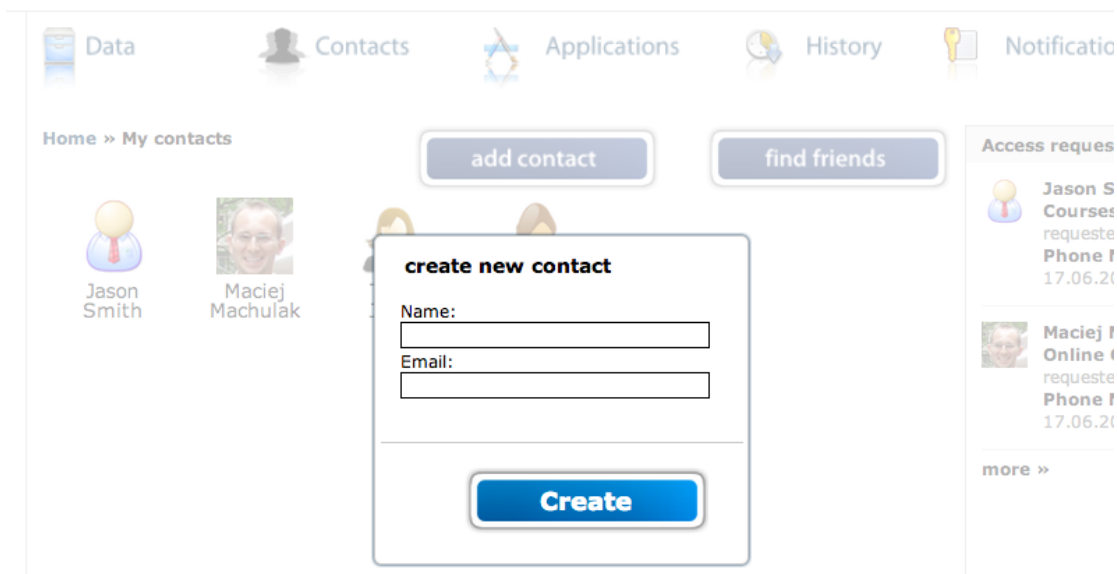


Figure 7.42: View showing the form that allows the user to add new contacts manually.

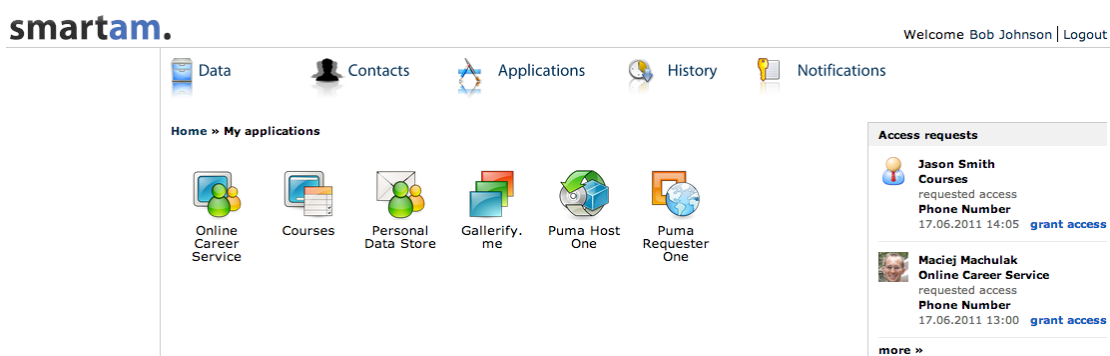


Figure 7.43: View showing the list of registered applications at AM.

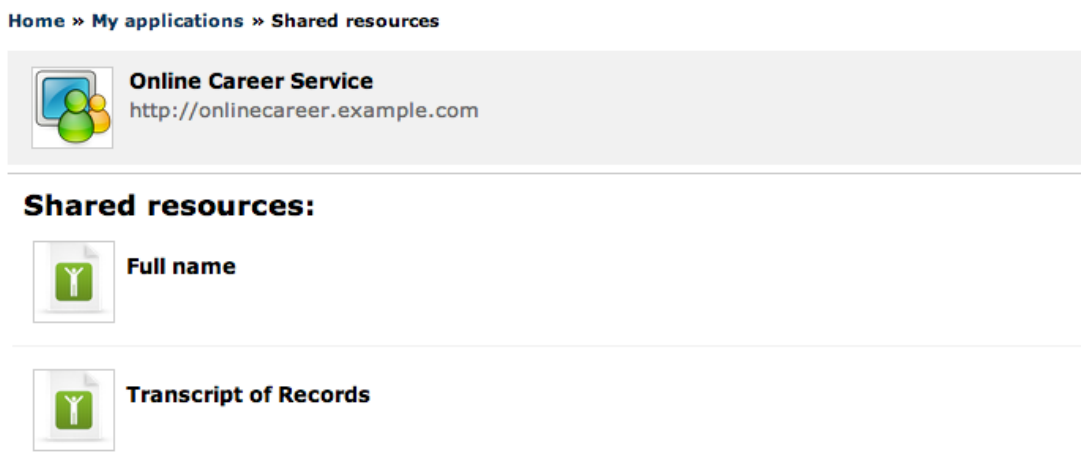


Figure 7.44: View showing resources shared with a particular application.

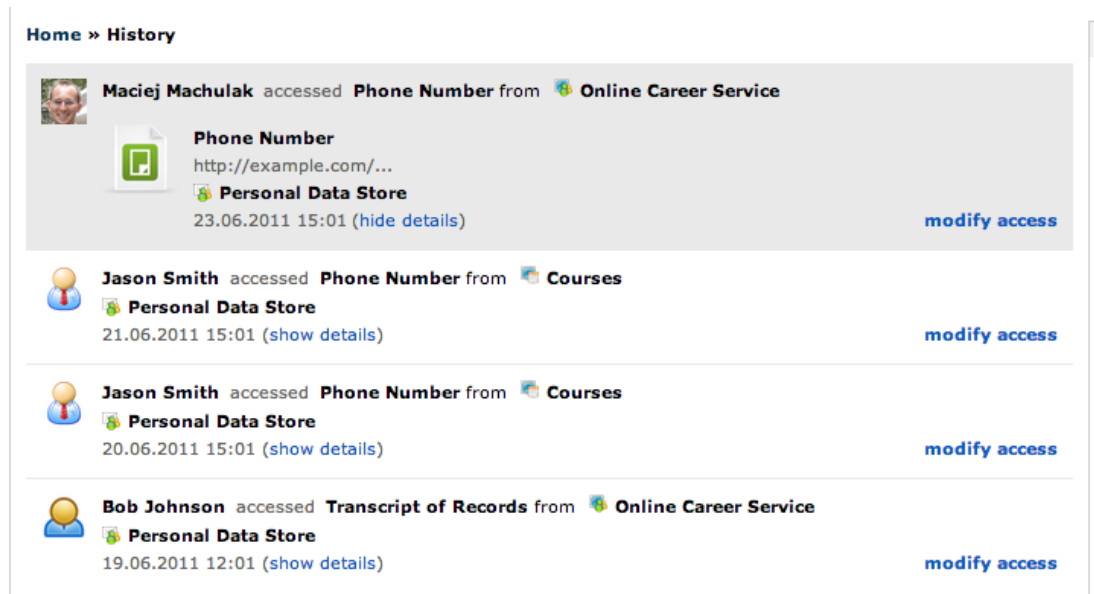


Figure 7.45: View showing access history list (audit log).

UI of SMARTAM V2 provides the user with a view for access history ("*History*") which is presented in Figure 7.45. The access history list displays each event of accessing a resource, with such details as the name of the person who has accessed it, the name of the application used for access, an icon of the resource, the name of the Host that stores this resource and a date when the resource was accessed. The user can easily amend the policy for resources and this is provided directly from the access history view (the user has to select the "*modify access*" option).

The UI also provides the user with a view for requests for access ("*Notifications*") which is presented in Figure 7.46. The request for access list provides information with pending inquiries

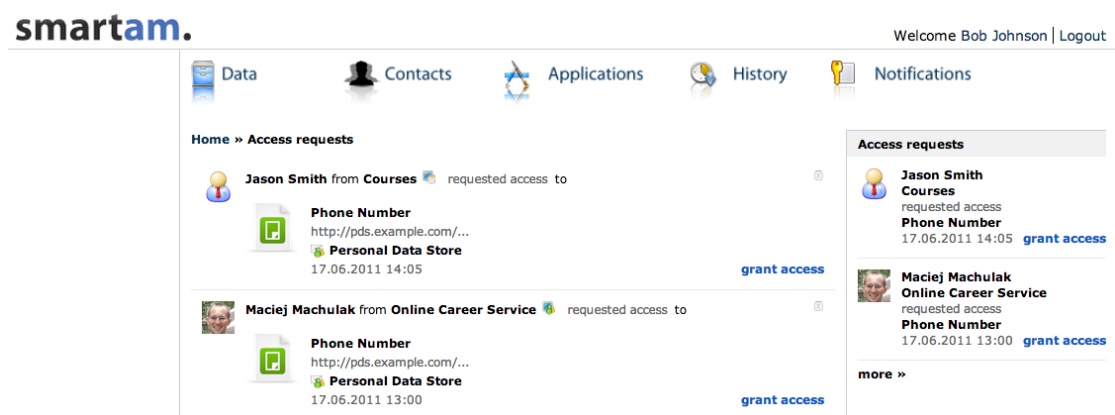


Figure 7.46: View showing requests to access protected data.



(a) View showing pending requests for access to data.

(b) View for granting access to data.

Figure 7.47: Request for access mechanism implemented by SMARTAM V2.

from Requesting Parties that wish to access protected resources. The same list is also visible for the user in the sidebar of the UI (see Figure 7.47). The user is given a choice to either accept such request or deny such request. When the user accepts such request then a new entry in the access control policy is created (Figure 7.47). Importantly, the AM sends an email to the Authorising User informing them about pending requests for access. When such requests are accepted, the AM may send the necessary notification to Requesting Parties (recall Figure 7.33).

7.4.8.3 Comparison

The newly designed User Interface addresses identified shortcomings and incorporates formulated requirements, which were presented in Sections 7.3.3.2 and 7.3.3.3 respectively. We visualise the summary of the list of most notable improvements to the AM User Interface in Figure 7.48.

In reply to the respondents' complaints of a counter intuitive arrangement of the layout and to address the shortcoming **SC1**, a new approach has been taken. Firstly, initiating the setup of an access control policy for a resource has been moved to the host application where a resource is located. The Host is integrated with an AM and can be provisioned with the location of a policy for resources that are registered for protection. Therefore, the Host can point the user directly to the actual access control policy for a resource. In particular, users do not have to visit the AM as such to protect their resources, but can start the flow of sharing resources directly from the Host.

Furthermore, the options available to the user on the main page of the presented AM have been changed to *"Data"*, *"Contacts"*, *"Applications"*, *"History"*, and *"Notifications"* and represent different sets of data available to the user. In particular, the user has direct access to one of the main functionalities provided by the AM, i.e. viewing the list of resources registered for protection and being able to access their policies (*"Data"* view). The user can also manage

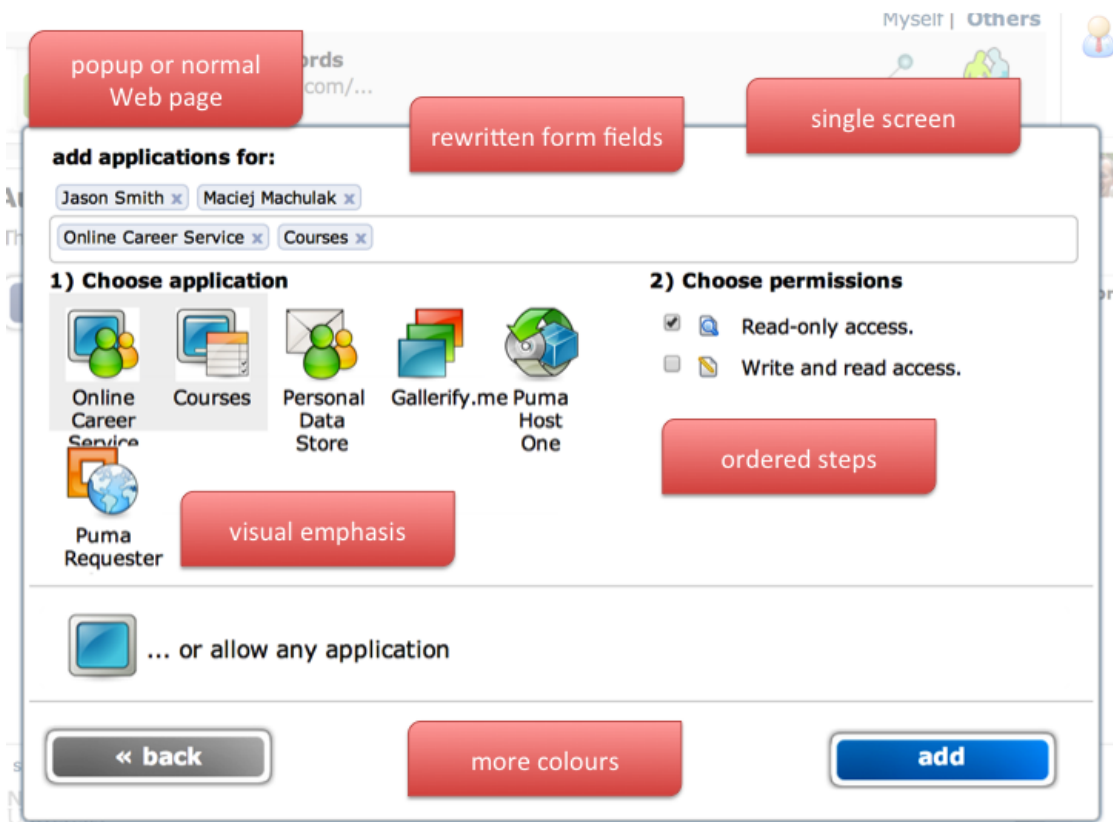


Figure 7.48: Visualisation of improvements to the User Interface introduced in SMARTAM V2.

their potential Requesting Parties and Requesters using the "*Contacts*" and "*Applications*" views respectively. Separate views are provided for displaying information about accessed resources ("*History*") and requests for access ("*Notifications*"). Setting up the policy for the purpose of sharing and protecting data has been simplified and has been provided with a wizard-like UI, as discussed in Section 7.4.8.3

Shortcomings **SC2** and **SC3** have been addressed by presenting the list of registered resources at the AM on the main screen, irrespectively of where these resources are located. It is envisaged that the user is able to filter resources, either by selecting the application or by providing few letters of the resource name in a search field, but the search feature has not yet been implemented. Each resource is associated with an icon, provided by the host application (or applied by AM, if the provided icon was missing), which allows the user to easily locate a resource and specify sharing settings for this resource. By clicking on the resource icon, a user is presented with a choice of policies, either for themselves or other users.

Nested lists have been introduced in exchange of the accordion module and drag-and-drop boxes, which were originally used in SMARTAM V1. This solution makes the form more direct and more interactive for the user. The form for setting up an access control policy resembles a wizard-like UI, which displays consecutive steps only after completing previous steps. The necessary form fields next to specific input fields have been provided. Furthermore, all labels have been rewritten and ordinal numbers for each field are now provided. This approach aims to guide users in the process of specifying access control policies for their online resources.

The design principle behind the newly designed User Interface for access control policies was to keep all the options in a limited number of screens. Also, features associated with a particular task should be displayed on a single screen. The goal was to display only features available at a particular time during the resource sharing task.

A truly single screen existed in one of the first revisions of the UI implemented for SMARTAM V2, which is presented in Figure 7.49. With the introduction of myself- and others-type policies, it was decided to provide separate screens for these two policy types. In the myself-type policy, a user is presented with a view of authorised applications and their associated permissions. When adding a new application, the user is presented with a simple screen where they can choose an application and authorised actions for this application. In the others-type policy, users are provided with the list of $\{Requesting\ Party, Requester, Actions\}$ tuples. When adding a new policy entry, users first choose the Requesting Parties. They can then restrict access to specific Requesters and select actions that should be authorised. This second step of adding applications and actions is initially hidden but is revealed when the user completes the first step of choosing Requesting Parties.

Home » My data » Sharing settings



All my videos

Resource url: <http://example.url>

make public

Permissions:

There are currently no permissions set.

add people

1) Choose contacts

Maciej Machulak x

Łukasz Moreń x

get contacts



Maciek
Wolniak



Maciej
Machulak



Luke
Kowalski



Łukasz
Moreń

2) Choose permissions

☐

Reading

☐

Editing

« back

Figure 7.49: Prototype view of UI for SMARTAM V2 allowing the user to specify access control policy in a single screen.

The aim was to provide a UI layout, which would make the user feel that they know which step they are at. Only after choosing contacts and applications/actions, the "add" call to action button is displayed. Therefore, the user cannot save sharing settings if all of the required information has not been provided. After the user successfully saves a policy, the view of this policy is reloaded asynchronously using AJAX without refreshing the page. The user can immediately see that the policy for a resource has changed and a new entry is available.

Furthermore, the identified shortcoming **SC4** of poor colour scheme has been addressed by introducing graphic icons next to each element of the top menu. Buttons and links have been redesigned to stand out more and to call user to action. Links in the text are coloured blue and when a user moves a mouse cursor over a hyperlink, then such link is underlined. The buttons are coloured deep blue, light blue and light grey. All major buttons are deep blue, large and in contrast to the background. The "add" button is the only one coloured light blue, in such a way it is distinctive and different than the rest of the buttons. Back button is coloured grey, as it is a less important option. All the confirmation buttons are coloured green and red. Negative answer buttons are coloured red and positive answer buttons are coloured green. It is believed that the proposed solution supports human ergonomics as it resembles the colours of the traffic lights with which most of the users are familiar with.

The help feature stated in the shortcoming **SC5** has not been introduced in the UI of SMARTAM V2. However, this UI is believed to be self-explanatory and any further assistance will be introduced only if required.

7.4.9 Implementation Details

SMARTAM V2 has been developed using the Spring Framework [74], which is a Java framework that provides a comprehensive set of tools for building multi-tier applications in Java. Spring is based on the *Inversion of Control (IoC)* and *Dependency Injection (DI)* design patterns [191]. Spring provides a convenient alternative to the Java EE architecture [34]. Instead of using EJBs (Enterprise Java Beans), Spring allowed to use simple POJOs (Plain Old Java Objects) where additional services (such as transactions or security, which were used extensively in the discussed implementation) were added to those objects by the application container. The container, through injecting specific dependencies, combines objects together based on a specific configuration.

Spring allowed to compose the presented software from individual components. In particular, it was possible to combine Discovery, OAuth 2.0, and UMA/j AM frameworks, among others, with components and services specific to the AM application. The aforementioned developed frameworks were tailored for Spring and were configured in Spring XML-based metadata

configuration files.

Spring Framework, in its current form, is based on a set of separate components that provide tools for solving specific development problems [74]. In particular, these tools allow developers to easily manage access to various relational and non-relational data stores, transactions, security or integration with external systems. These components include the following modules: core container, data access/integration modules, Web, Aspect-Oriented Programming (AOP), instrumentation and tests. The core container manages dependency injection and provides the inversion of control functionality. It manages object lifecycle and provides generic functionality to be used by other modules (e.g. event handling or application context management).

In Spring, data access and integration modules allow the application to communicate with databases using JDBC [33] or such tools as Hibernate that rely on *ORM* techniques. In Spring, the Web layer is provided through the Spring Web MVC Framework [76] that simplifies development using the Model-View-Controller (MVC) design pattern [209]. Aspect oriented programming (AOP) allows Spring to separate business logic from other non-business related functionality, such as transactions or security management. The test module, on the other hand, simplifies application testing with such popular tools as JUnit or TestNG and provides support for mock objects (allowing isolated testing of applications).

7.4.9.1 Presentation Layer

The SMARTAM V2 UI has been implemented according to the MVC design pattern. The Spring Web MVC framework has been used for this purpose (Figure 7.20). As discussed, requests for specific views of the discussed AM (e.g. resources, contacts, etc.) are routed by implemented backend *View Controllers*, which operate on *Models* and return appropriate *Views*. Importantly, SMARTAM V2 provides very lightweight controllers only and in most cases these controllers are required to return a view and not provide any sophisticated operations on the model. Instead, SMARTAM V2 leverages the AJAX-based communication between the view that is returned to the user's Web browser and the backend API.

A set of JavaScript functions has been developed to support communication between the browser and backend controllers. These functions have been developed using the jQuery library [36] and can request the data from the AM's AJAX API and generate results in the HTML format. The Pure templating tool [67] has been used to generate HTML from JSON data. The Amazon S3 has been used as the Content Delivery Network. This service has been used to host most of CSS and JavaScript files, as well as such static files as images.

UI of SMARTAM V2 follows the rules of the *Responsive Web Design*, which allowed to implement the system to be easily accessible from a range number of different devices, including

smartphones and tablets. The latter ones, in particular, were becoming more common and the UI has been adapted to be accessible on Apple iPad tablets, too.

7.4.9.2 Services Layer

The business logic has been implemented as services and components of the Spring framework. Separate services are provided for each core functionality related to users, host and requester applications, resources and their associated policies, etc. Services expose both fine-grained and coarse-grained operations on resources. The Spring-provided transactions are used to provide ACID properties [197] for these operations. Developed services are injected into Web controllers and other services, and allow to operate on the data provided by the AM. Importantly, these services do not access data directly but use implemented DAOs for this purpose.

7.4.9.3 Persistence Layer

SMARTAM V2 uses a MySQL Community Edition relational database [132] to store information about users, resources, access control policies, and other data. SMARTAM V2 does not have direct access to the database but leverages the Hibernate framework and the *Hibernate Query Language (HQL)*, which provide ORM and allow to easily map objects to database tables. In order to minimise the number of database access requests, SMARTAM V2 uses the Infinispan framework [31], which caches the data received from the database in an efficient No-SQL key/value datastore and makes it available to services of the AM.

7.4.9.4 Application Programming Interface

API endpoints have been implemented using the Apache CXF framework [138] and its JAX-RS component [271]. This framework has been used for both AJAX-based as well as UMA-related endpoints. The developed endpoints are designed to be RESTful to the maximum possible extent and are based on existing HTTP methods.

Developed frameworks, such as Discovery, OAuth 2.0, and UMA/j AM, have been developed with support for Apache CXF. Therefore, these frameworks have been added to the AM with minimum effort. In most cases, it was only required to provide configuration of specific endpoints in Spring XML-based configuration files and to implement necessary interfaces of data providers or validators of these frameworks.

The UMA-related functionality in SMARTAM V2 has been implemented using UMA/j Authorisation Manager framework, which provides implementation of *Protection API* and *Authorisation API*. UMA/j AM has been built as the first known framework to provide developers with the ability to implement custom UMA-compliant Authorisation Managers. UMA/j ensures

that API endpoints exposed by AMs provide functionality that conforms to the UMA protocol. Importantly, it provides these endpoints as Java Servlets as well as JAX-RS endpoints, which supports flexibility in building Web applications.

Existing API endpoints provided by this framework were included in SMARTAM V2. UMA/j AM requires configuration mostly and allows such configuration to be specified in Spring XML-based metadata configuration files. Furthermore, various custom data providers and validators, which were necessary for different API endpoints, have been implemented. For example, adding the *Resource Registration* endpoints to SMARTAM V2 required including the `JAXRSResourceRegService` from UMA/j and implementing the CRUD-like `ResourceRegProvider` interface that is required by this service.

7.4.9.5 Security

SMARTAM V2 extensively uses Spring Security [75] - a framework that provides out-of-the-box user authentication and user authorisation functionality for Web applications. Spring Security allows to manage user sessions at the AM, irrespectively of how the user got authenticated in the first place (e.g. it was easy to use the described Facebook login instead of traditional username/password login). Additionally, it provides access to a *Security Context*, which is associated with each session, which is used during access control of users at a method level. For example, the user identifier is checked for authorisation at various layers of the AM, including controllers as well as services.

Spring Security has been extended with support for the OpenID Connect, OAuth 2.0 and UMA protocols by implementing these protocols as additional modules. With this approach, users can authenticate to SMARTAM V2 with their existing identities. Furthermore, client applications can access various APIs provided by SMARTAM V2 and UMA-compliant applications can interact with Protection API and Authorisation API of this AM.

7.4.10 Limitations

SMARTAM V2 was the second implementation of an UMA-compliant Authorisation Manager that addressed various limitations that were identified in SMARTAM V1. However, during the research and development of the UMA protocol further shortcomings have emerged and some of these shortcomings have affected this AM implementation.

This section discusses the following limitations of SMARTAM V2:

1. Access control policies are not reusable;
2. Limited support for claims-based policies;

3. Limited support for OpenID Connect;
4. Support for bearer tokens only;
5. Limited grouping of resources at AM;
6. Limited support for notification management;
7. Lack of support for resource types;
8. Lack of support for negative policies;
9. Access to resources via UMA-enabled applications only.

Unlike the first AM implementation, SMARTAM V2 does not currently support reusable policies. The Authorising User cannot define a policy and then apply it to numerous resources. Instead, the user has to reapply the policy at the central AM (this already introduces a level of convenience for the Authorising User). Although the lack of reusable policies can be considered as a significant limitation, it can be easily resolved and this is planned in further versions of the software. In particular, SMARTAM V2 can be extended to support policy templates. Support for such templates would allow the user to create a policy for a particular resource and then save it as a template. This template could then be applied to other resources as necessary.

SMARTAM V2 supports only redirect-type claims. Such claims require the Requesting Party to be redirected by the requester application to this AM. The Requesting Party then engages in the claims gathering flow, either by signing in with their account or by providing OIDC claims to the AM. Support for OIDC claims is currently limited and allows the Requesting Party to only assert to the AM that they are the owner of a particular email address. This limitation is related to the fact that SMARTAM V2 currently uses only the `email` field of the supplied OIDC claims. Therefore, implemented access control policies for resources may define that only specific users (owners of a particular email address) will be given certain permissions for a resource/service. This behaviour is different from SMARTAM V1 where Authorising Users could define claims that would need to be self-asserted by Requesting Parties, irrespectively of the identities of those parties.

Tokens issued for Requesters are bearer-type tokens and this is similar to SMARTAM V1. This requires Requesters to carefully handle such tokens and not expose them to malicious third parties. Moreover, these tokens are opaque to Hosts. Therefore, a Host has to refer to SMARTAM V2 to check every access request that comes from a Requester. Unlike in SMARTAM V1, the SMARTAM V2 informs a Host about the permissions associated with a token. Therefore,

a Host can reuse this information for subsequent access requests that include the same token from a Requester.

Hosts can register resources for protection at an AM and can group them as required through the notion of resource sets (recall Section 5.4.2). However, these resource sets cannot be further grouped at the SMARTAM V2. It would be beneficial to allow the Authorising User to register numerous resources from various host applications and then group them if these resources are logically connected (e.g. recall the scenario of applying for a job position that requires sharing of resources from a set of distributed Hosts). A single access control policy could then be applied to such a group instead of to individual resources.

As discussed in Section 7.4.5, when an Authorising User creates an entry in an access control policy then a notification is sent to a Requesting Party who may be concerned with accessing a resource/service. An Authorising User currently cannot control which information is sent and this could be introduced in further versions of the software. For example, an Authorising User could decide when an AM sends emails or other notifications.

When a resource is first registered at an AM, it is associated with an empty access control policy (i.e. there are no sharing setting entries associated with this resource). It is up to an Authorising User to create new entries in that policy and share a resource. This limitation in terms of user experience could be addressed by allowing policies to be automatically assigned to resources based on resource types. For example, if a user registers social-type resources (e.g. videos, photos, blog entries, and similar) then these resources could be potentially shared based on a well-defined logic at an AM (e.g. social-type resources could be by default shared with all friends of an Authorising User). It would be important, however, to inform a user about the access control policy that would be applied to a resource during its registration and before that resource is actually shared.

Authorisation Managers are envisioned for managing access control for various resources stored on distributed hosting applications. With vast amount of information, it would be necessary to support user with any policy management functions, such as those mentioned in Section 2.6.2.3. SMARTAM V2 is currently limited in this respect. In particular, this AM does not allow for resource types to be used and such resource types could simplify management at the AM and would also help to visualise for an Authorising User what data is protected and in which way. Importantly, further tests regarding the use of SMARTAM V2 that provides its functionality for various hosting applications and large amounts of different resources should be conducted.

Similarly to SMARTAM V1, the second AM implementation does not support negative policies. Therefore, certain use cases and scenarios may need to be implemented using more complex positive policies. Requesting Parties have to be given explicit permissions to access

resources and it is not easily possible to give them access to all resources and exclude only certain ones.

Resources and services protected with SMARTAM V2 can be accessed by Requesting Parties via requester applications that understand the UMA protocol. As such, typical Web browsers cannot act as clients unless these browsers are equipped with some plugins that are able to interact with a Host and an AM according to the UMA protocol. This is considered as one of the limitations of the implemented AM.

7.5 Chapter Summary

This chapter presented two UMA Authorisation Manager implementations. It started by discussing an initial AM implementation, called SMARTAM V1. It presented the design of this software and its policy model. It also discussed integration of SMARTAM V1 with example applications.

This chapter then presented the conducted user study that was used for evaluation of SMARTAM V1 and its User Interface. It presented identified shortcomings of SMARTAM V1 based on the gathered feedback from the user study. It also presented requirements that were derived from these shortcomings and that were used in further work on UMA software.

Moreover, this chapter discussed a second AM implementation, named SMARTAM V2. SMARTAM V2 addresses various shortcomings that were identified in the earlier implementation. It also provides support for a more recent revision of the UMA protocol. This chapter discussed the architecture of this software and its policy model. It also presented its unique features that provide additional value on top of the core UMA proposal. This chapter also provided an overview of shortcomings of SMARTAM V2.

Chapter 8

Conclusions

This chapter concludes the presented work. It first summarises the thesis. This summary is followed by a consideration of future research directions and concluding remarks.

8.1 Thesis Summary

This thesis pointed out the weaknesses in existing systems managed by different authorities. It also discussed the requirements for a new system that would address these weaknesses. This thesis discussed a novel authorisation system called User-Managed Access Control. It also presented the User-Managed Access (UMA) proposal that is researched by the User-Managed Access Work Group at Kantara Initiative. The proposed systems, unlike existing ones, allow the user to play the pivotal role in the workflow of sharing data securely and efficiently with other users and services on the Web. This thesis also presented details of the developed UMA software.

Firstly, Chapter 1 introduced the thesis. It outlined the main goals and the approach taken in satisfying these goals. It also introduced the motivation behind a new access control solution for distributed and user-driven environments such as Web 2.0. It presented an example scenario describing a transaction on the Web. It provided analysis of this scenario and pointed out the shortcomings of existing access control systems. It then presented formulated requirements for a new access control proposal. This chapter summarised the novelty of the solutions presented in this thesis and also discussed main contributions.

Access control in distributed environments was presented in Chapter 2. This chapter provided background information regarding concepts that were used throughout this thesis. It started with an overview of multi-domain computing environments, the open Web and Web 2.0 applications, and presented their specific characteristics that are of interest in the context of

access control. Furthermore, it discussed different concepts of access control in distributed environments, access control policies as well as delegated authorisation. It then discussed challenges in access control policies and architectures.

Chapter 3 provided a literature review. It discussed existing standards and standard proposals as well as academic work. It presented various solutions for access control that allow individuals to be in control of access to their Web data.

Chapter 4 introduced a novel authorisation solution for distributed Web resources, called User-Managed Access Control (UMAC). It discussed the architecture and the protocol of this proposal. It also showed how this proposal meets identified requirements for a new user-managed access control system. It then discussed the identified shortcomings and limitations of UMAC.

The UMAC approach puts a user in full control of access to their resources on the Web. It relies on a user's centrally located security requirements for those resources, which may be scattered across multiple distinct Web applications. These security requirements can be expressed in the form of access control policies and are stored and evaluated in a specialised user-chosen component. Applications delegate their access control function to this component.

Chapter 5 presented the User-Managed Access (UMA) solution, which is researched by the User-Managed Access Work Group at Kantara Initiative. UMA has been used in research presented in this thesis. This chapter discussed additional requirements for UMA. It also presented the UMA architecture and the protocol. This chapter then evaluated UMA and provided an overview of its limitations. It also showed differences between UMA and the UMAC approach.

UMA provides a method for users to control third-party access to their protected resources, residing on any number of host sites, through a centralised authorisation manager that makes access decisions based on user instructions. UMA gives users the required flexibility in sharing their data and supports requesters with accessing such protected data, similarly to the previously discussed UMAC proposal.

Chapter 6 presented two UMA frameworks that allow applications to externalise their access control functionality and act as Hosts and Requesters, as defined by the UMA protocol. Section 6.2 presented a Java implementation named UMA/j. Section 6.3 presented a Python implementation named Puma. This chapter also discussed identified limitations of both frameworks.

UMA/j was targeted at enterprise-level (but still user-driven) cloud-based Web applications. The framework provides a high-level *UMA API* for applications as well as allows to use low-level APIs for finer control of the protocol flows.

PUMA, on the other hand, was built primarily for applications running on the Google App Engine PaaS. Because this framework was designed and implemented later than UMA/j, it contained various enhancements and simplifications for application developers. It also implemented

a more recent revision of the UMA protocol.

Chapter 7 presented two UMA Authorisation Manager implementations that were developed in parallel to the work on UMA frameworks. It started by discussing an initial AM implementation, called SMARTAM V1. It presented the design of this software and its policy model. It also discussed integration of SMARTAM V1 with example applications.

SMARTAM V1 allows a user to compose access control policies and apply them to a set of resources hosted on UMA-enabled Web applications. These applications delegate access control to this AM and are only concerned with enforcing access control decisions. The AM provides a User Interface for managing policies and a RESTful API for client applications.

Chapter 7 then presented the conducted user study that was used for evaluation of SMARTAM V1 and its User Interface. It presented identified shortcomings of SMARTAM V1 based on the gathered feedback from the user study. It also presented requirements that were derived from these shortcomings and that were used in further work on UMA software.

Moreover, Chapter 7 discussed a second, improved AM implementation, named SMARTAM V2. SMARTAM V2 addresses various shortcomings that were identified in the earlier implementation. It also provides support for a more recent revision of the UMA protocol. This chapter discussed the architecture of this software and its policy model. It also presented its unique features that provide additional value on top of the core UMA proposal. This chapter also provided an overview of shortcomings of SMARTAM V2.

Similarly to the first version, SMARTAM V2 allows the user to compose access control policies and apply them to a set of resources hosted on different Web applications. These applications delegate access control to this AM and are only concerned with enforcing access control decisions. SMARTAM V2 allows requester applications to obtain authorisations to access protected resources.

Finally, Chapter 8 concludes this work. It discusses a summary of the thesis. It also presents future research directions and finally provides concluding remarks.

8.2 Future Work

The main future research directions are:

1. Deployment and acceptance of user managed access control approaches;
2. Development of improvements to UMA protocol:
 - (a) Optimised integration with client applications;
 - (b) Schema for protected resources;

- (c) Protected resources discovery support;
 - (d) User's preferred AM discovery support;
3. Development of improvements to SMARTAM implementation:
- (a) Extended audit with support for arbitrary events;
 - (b) Extended policy support;
 - (c) Visualisation of complex policies;
 - (d) User studies of the newly implemented AM;
 - (e) Integration of AM with federated authentication proposals;
 - (f) Mobile applications support.

These directions are discussed in further sections.

8.2.1 Deployment and acceptance of user managed access control approaches

Despite numerous advantages of such solutions as UMAC or UMA in particular, as well as the presented UMA implementations, it is still too early to tell if users will fully appreciate the new functionality. In particular, it would be necessary to focus on broader deployment and integration of the UMA protocol with existing applications used on the Web. Such integration will allow for real-world validation of the proposed solution and its applicability to protecting distributed Web resources.

There are various use cases where it has been identified that the individual should be empowered to control access to the data. Some of these use cases have been submitted to the UMA WG and are available at [85]. New implementations supporting similar scenarios have been emerging. By allowing the individual user to be in control over their data, the user can make better choices how such data is shared and used, which may potentially provide a better value not only to the user but also to the recipient of the data. However, user uptake is still unpredictable, and it will be very interesting to find out to which extent users will utilise the features of user managed access approaches. In particular, it will be beneficial to investigate how UMA fits into existing business models (or changes them) where users are offered a free service and in exchange the application gets access to the personal data of the users.

Furthermore, the UMA approach should be aligned with other existing access control proposals that should increase its adoption. Such solutions as Google's Street Identity can be perceived as the first approach towards "*UMAnising*" APIs and allowing for more user-controlled access to distributed resources. Further work is necessary on making the UMA protocol a real standard, published as RFC at IETF by the User-Managed Access Work Group.

8.2.2 Development of improvements to UMA protocol

This section discusses potential future work in the area of UMA protocol improvements.

8.2.2.1 Optimised integration with client applications

Users of host and requester applications are required to go through the complete OAuth 2.0 flow, which is a core part of the UMA proposal, in order to authorise these applications to use functionality provided by authorisation managers. This flow is required for applications to obtain HAT or RAT tokens respectively. Importantly, the user has to go through such a flow (or a similar one) twice in case their application uses federated authentication, i.e. once during the sign in with their preferred IDP and once during application authorisation for their preferred AM.

It would be beneficial to simplify the process of obtaining HAT and RAT tokens by applications. In particular, it should be possible for applications to obtain the required token when the user signs in to such an application. This would allow the user to sign in to the client application (either host or requester) and authorise this application for the AM functionality in a single flow and with a single redirect only. For example, when the user signs in to the AM with the OpenID Connect protocol, then the client application would, apart from user information, obtain the HAT/RAT token for a particular AM that the user chooses to use.

8.2.2.2 Schema for protected resources

Current UMA proposal allows host applications to register arbitrary resources at the user's preferred Authorisation Manager. This registration requires certain information to be provided by the host application, including the name of the resource as well as its icon URI. Such information is used mostly to support the user during policy management steps at AM and is required at the User Interface level.

Work is planned on providing a schema for most commonly used data types which could be protected using UMA. Host applications would be able to register resources and specify their type rather than their icon URI or even their name. This solution would achieve two goals.

Firstly, the user experience would be standardised at the AM by allowing the AM to decide on the icons to be displayed based on the particular resource that was registered. This will allow the user to more easily review their protected resources by their specific type. For example, it will be possible to identify personal information, documents, or photos based on their category. Such user experience is of particular importance on the open Web where users are concerned with various data types distributed among numerous applications and the registration of their

various resources at a single AM.

Secondly, it would be possible to make a step forward towards allowing the AM to associate existing policies to newly registered resources without any user intervention. In particular, a user that registers a resource should not be concerned with manually associating a policy to protect and share this resource. Instead, the Authorisation Manager could suggest a policy that should be applied based on the resource type.

Moreover, by applying categories to different resources, the user will be able to review security settings for those resources more precisely, either manually or with the help of the AM. For example, the user might be provided with a view of policies for their personal information and will be able to quickly detect any anomalies. Another example is where the AM will be able to notify the user that a policy for a resource of a particular type is different from policies for other resources of the same kind.

8.2.2.3 Protected resources discovery support

The UMA proposal does not specify how requester applications learn about protected resources that these applications could access for a particular user. For example, a user of an online job application system must provide the location of their PDS that could be used as a source of data, in case this system integrates with numerous PDS applications. Furthermore, an online job application system is unable to detect if the PDS actually stores a required piece of information.

It would be beneficial to allow the requester discover applications used by the user for specific types of protected resources. For example, the requester could discover applications used by the user for storing verified phone numbers or "Transcript of Records" documents. The user would then be presented with a limited set of choices when trying to import the data to their requester application. Such discovery would also allow requesters to provide additional functionality if the requester can discover that the user has a particular resource type. For example, the application can offer *premium-type* functionality in case the user decides to provide it with a verified phone number. Such solution is adopted by the Google's Street Identity protocol that defines a discovery endpoint.

8.2.2.4 User's preferred AM discovery support

UMA does not specify how host applications can discover the preferred AM of the user dynamically. In particular, the user is concerned with choosing the right AM after signing in to the application. The user can be presented with an input field where they must provide the URL of the AM or they can be provided with a UI that supports them with the AM selection process. In the latter case, the user might be challenged with numerous logos of potential authorisation

managers and this is commonly referred to as the *nascar problem*.

It would be useful to provide hosts with the ability to discover the user's preferred AM during the sign in process. In particular, such discovery would be possible with authentication federation protocols. The user's preferred AM could be provided as one of the attributes available for the user at their IDP (e.g. as one of the claims returned by the `UserInfo` endpoint available in OpenID Connect). Moreover, the AM function could be added on top of the IDP, which would eliminate the need for such discovery. For example, the application would be provided with a token for the IDP and AM during the sign in process. In fact, such approach of combining identity providers and authorisation managers has been already discussed and considered by the UMA WG.

8.2.3 Development of improvements to SMARTAM implementation

This section discusses potential future work in the area of AM development.

8.2.3.1 Extended audit with support for arbitrary events

SMARTAM V2 allows users to quickly review how their resources are accessed by requesting parties and requesters. This feature is provided using a separate UI view. This view presents a list of entries representing successful access requests to protected resources.

However, this view currently supports access requests to UMA-protected resources only. This view is planned to be extended with support for other events of the AM. In particular, this view could include policy specification related events (e.g. creation or deletion of a new entry in a policy for a particular resource) or resource registration/de-registration events. Different approaches to showing this information to the user are planned to be investigated, e.g. based on the level of events, which is similar to adopted logging strategies in software development.

8.2.3.2 Extended policy support

SMARTAM V2 provides support for myself- and others-type policies. However, a single policy cannot be reused for multiple resources. Reusable policies were only provided in SMARTAM V1. The existing SMARTAM V2 implementation is planned to be extended to allow users to define *policy templates* which could be applied to newly registered resources. For example, the user could share their newly uploaded photos with a predefined set of friends for whom a policy has been created earlier, or with a predefined set of services.

The policy model is also planned to support arbitrary third-party asserted claims. Firstly, it would be necessary to support OpenID Connect claims in policies. SMARTAM V2 implementation supports only the `email` attribute and allows users to present a verified email to get access

to protected resources. Future plans include extending this support with other claims that are specified in [57].

Furthermore, it might be useful to allow users to compose their own contracts. These contracts would be enforced by services accessing data. Contracts are particularly important in *Person-to-Service* sharing scenarios. For example, a user could share their data only for a specific purpose and under specific terms, which is one of the goals of the UMA proposal. The purpose and the terms could be defined in a document. Adherence to terms would be acknowledged by the receiving party and this would have legal enforceability.

An interesting use case for policies is also presented in [243]. This use case discusses monetisation of access to resources. It would be possible for the user to use UMA to monetise access to their data even if the hosting application did not support such feature. During access to a protected resource, the requesting party would be required to provide a payment confirmation and only then would be provisioned with the necessary permissions for the RPT token.

8.2.3.3 Visualisation of complex policies

UMA centralises authorisation policies for user's distributed Web resources. The externalisation of policies from host applications introduces a new level of complexity for the user. In particular, the user has to map the authorisation structure for their different resources and with numerous resources and their policies in place, this might be a difficult task. It is likely that the user will be lost with too many policies in place and will not be provided with a meaningful holistic view of the security settings for their online data. This may result in the loss of privacy, exploitation of online personal information or a security breach if data is not shared as intended [155].

Research should be continued in providing the user with a better understanding of sharing of their data in such highly distributed environment as the open Web. Importantly, policy visualisation techniques could be used to enhance privacy controls available at the developed AM. Initial attempts towards achieving this goal have been already made with preliminary research conducted as a joint effort with Domenico Catalano. Initial presentation of this research can be found in [155]. This research aims to define a context of the data sharing policy. A visualisation approach is planned to be used to help users define the appropriate context of sharing their information. A visualisations of the proposed concept, which is named UMA Connection, is shown in Figure 8.1.

8.2.3.4 User studies of the newly implemented AM

During the presented research, the functionality and benefits provided by UMA as well as the User Interface of SMARTAM V1 were evaluated. Further evaluation of SMARTAM V2 should

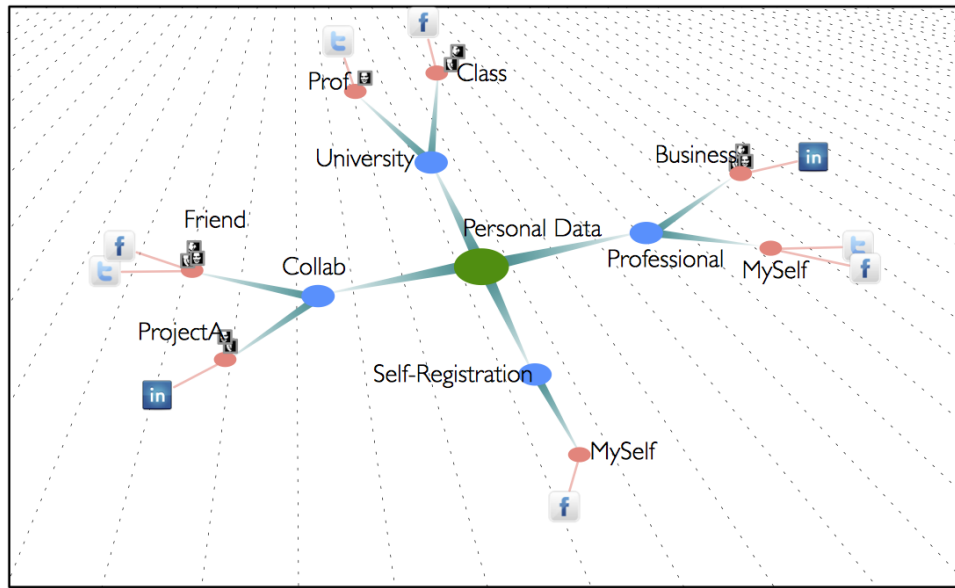


Figure 8.1: Visualisation of UMA Connection concept [155].

be also conducted in order to gather feedback on the newly designed UI of this AM. The goal is to investigate the newly proposed policy model that allows users to share their resources using myself- and others-type policies. Furthermore, it would be necessary to investigate how users respond to the new functionality introduced in SMARTAM V2, i.e. access history as well as request for access notifications.

8.2.3.5 Integration of AM with federated authentication proposals

SMARTAM V2 proposal could be integrated with other third party Identity Providers using such novel protocols as OpenID Connect. OIDC is one of the emerging Web technologies that aims to solve the problem of authentication on the Web. Users could be able to sign in to the AM with their existing accounts from their chosen OIDC-enabled IDPs. This would significantly simplify the login user experience to the AM.

However, it would also pose interesting challenges with regard to policy specification. For example, a user can share a resource with their friend based on this friend's Google identity but such friend can sign in to the AM with their account from another OIDC-enabled IDP when trying to satisfy the policy. In such situation, the policy would not be satisfied and access to a resource would not be granted. Account linking is therefore necessary and such linking would allow a user to link multiple federated identities to their local account at the AM. This issue is planned to be investigated further.



Figure 8.2: User Interfaces of a prototype mobile application integrated with SMARTAM V2.

8.2.3.6 Mobile applications support

To allow building mobile applications that could integrate with the AM system, a *Management* endpoint in SMARTAM V2 was designed. However, support for mobile is in its very early stages of development. Further development efforts are planned and this would allow other developers to be able to build applications on top of this AM. In particular, the goal is to expose the policy specification functionality over an API. With this approach, others will be able to build their own user interfaces and still benefit from the proposed rich policy model as well as already implemented UMA functionality,. Figure 8.2 illustrates the UI of a prototype implementation of a mobile application that supports viewing audit log information. This prototype application was implemented using the Feedhenry framework [12].

8.3 Concluding Remarks

The open Web environment has emerged as an ubiquitous platform that allows numerous online transactions to take place. Web applications have become more user-centric and allow their users to create different types of data, and to disseminate this data and share it with other users and services. With the ever-growing number of resources owned by individual users, there is a clear need for new ways to control the data and its access by other users or services. The proposals discussed in this thesis aim to provide a solution to protect resources according to user's requirements. Supporting users with making meaningful choices regarding their data sharing is an important step in realising one of the goals of the Web - making the user the core part of this platform.

Appendices

Appendix A

Graphical representation of UMA/j module dependencies

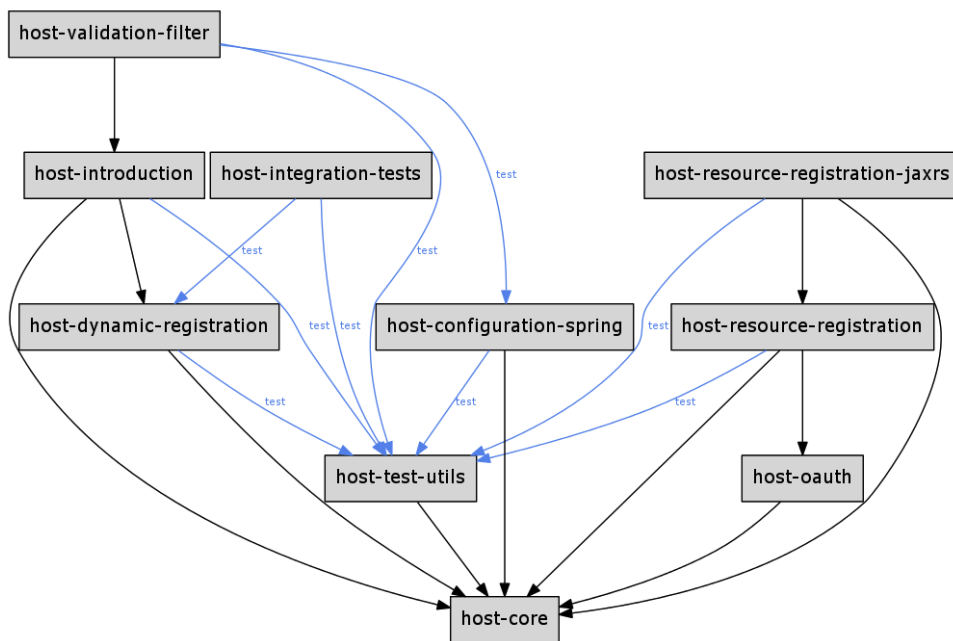


Figure A.1: UMA/j Host module dependency graph.

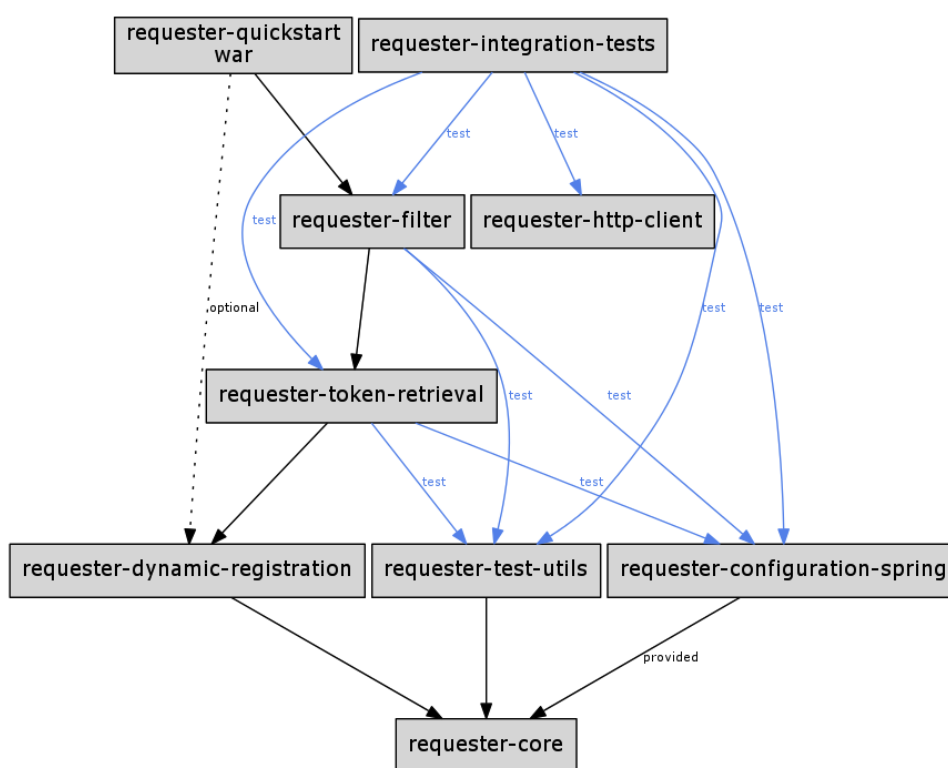


Figure A.2: UMA/j Requester module dependency graph.

Appendix B

Example UMA/j Host configuration file

```
1 <beans>
2   <jaxrs:server id="uma" address="/uma">
3     <jaxrs:serviceBeans>
4       <ref bean="resourceRegEntryService" />
5       <ref bean="deleteService" />
6     </jaxrs:serviceBeans>
7   </jaxrs:server>
8   <bean id="umaValidationProvider"
9     class="...HttpUMAHostValidationProvider">
10     <property name="accessDecisionEvaluator"
11       ref="gallerifyAccessDecisionEvaluator"/>
12     <property name="requester" value="false"/>
13   </bean>
14   <bean id="abstractResourceReg"
15     class="...AbstractResourceRegistrationOAuthAwareService"
16     abstract="true">
17     <property name="provider" ref="gallerifyUMAProvider"/>
18     <property name="oauthAuthorizationUrl" value="{oauth.entry}"/>
19   </bean>
20   <bean id="resourceRegEntryService"
21     class="...ResourceRegEntryService" parent="abstractResourceReg">
```

```
22 </bean>
23 <bean id="deleteService"
24     class="...ResourceRegDeleteService" parent="abstractResourceReg">
25 </bean>
26 <bean id="resourceRegExceptionMapper"
27     class="...ResourceRegExceptionMapper"/>
28 <bean id="gallerifyUMAProvider" class="...GallerifyUMAProvider">
29     <property name="actions">
30         <list>
31             <bean class="net.smartam.uma.host.core.domain.UMABasicAction">
32                 <constructor-arg name="actionId" value="view_album_api"/>
33                 <constructor-arg name="actionUri"
34                     value="http://static.gallerify.me/json/view_album_api.json"/>
35                 <constructor-arg name="actionName"
36                     value="Viewing albums from external applications."/>
37                 <constructor-arg name="actionIconUri"
38                     value="http://static.gallerify.me/images/api.png"/>
39             </bean>
40             <bean class="net.smartam.uma.host.core.domain.UMABasicAction">
41                 <constructor-arg name="actionId" value="view_album"/>
42                 <constructor-arg name="actionUri"
43                     value="http://static.gallerify.me/json/view_album.json"/>
44                 <constructor-arg name="actionName"
45                     value="Viewing albums in Gallerify.me application."/>
46                 <constructor-arg name="actionIconUri"
47                     value="http://static.gallerify.me/images/read.png"/>
48             </bean>
49         </list>
50     </property>
51 </bean>
52 <bean id="umaProperties" class="...PropertiesFactoryBean">
53     <property name="locations">
54         <list>
55             <value>classpath:uma-config-host.properties</value>
56         </list>
```

```
57     </property>
58   </bean>
59 </beans>
```

Listing 25: Example UMA/j Host configuration specified in the Spring XML-based configuration metadata file.

Appendix C

UMA/j Implementation and Evaluation

C.0.1 Implementation

UMA/j has been implemented in Java according to software engineering best practices with design patterns applied where necessary. Its modularity allows for adjustment to different deployment environments and the use of its different components independently. For example, the initially embedded OAuth 2.0 module has been externalised and can be now used as a standalone library (see Section 6.2.4 for more details). The same applies to Discovery and Dynamic Introduction modules. Moreover, UMA/j does not rely on any specific Web framework and can be easily integrated with applications that conform to the Java Servlet specification [101].

Unit tests were provided for individual components. The framework's compliance with the UMA protocol¹ was verified using integration tests. The project can be built using Apache Maven [47]. Therefore, its modules can be added as dependencies to software that uses Maven as a build and packaging manager.

C.0.2 Evaluation

UMA/j has been evaluated empirically from the perspective of Web application developers as well as end-users of these applications. Performance or scalability tests have not yet been conducted. UMA/j Host was tested empirically with two Web applications - an online storage service and an online photo gallery service that can be deployed to the Google App Engine (GAE) platform. Both applications provide User Interfaces and RESTful APIs. The online storage service is

¹As discussed in this thesis, the UMA protocol was changing at the time this framework was developed. Therefore, compliance was tested against an existing revision of UMA at the time of framework development.

based on Java servlets, while the online gallery service is based on the Spring MVC framework. RESTful APIs are provided using Apache CXF. UMA/j was integrated with both applications and tested using SMARTAM V1.

Integration of UMA/j Host required some programming effort and changes in configuration files of the implemented applications. Firstly, necessary dependencies were included in the classpath of these applications (since UMA/j is provided as Maven artifacts, it was possible to add the framework as a dependency to the Maven `pom.xml` file of each of example applications). Then, the UMA API was added to provide a clear separation between the framework and the business logic. Discovery, dynamic registration, introduction (OAuth 2.0), and request filtering were configured by adding software modules to deployment descriptors of Web applications (i.e. in `web.xml` files). The first three modules were included as a single servlet, while the last module was included as a filter. Configuration for all UMA/j modules was provided in an XML configuration file and was included in WAR files of both applications. UMA functionality was exposed to end-users with a custom UI at each application.

End-users of implemented Web applications were able to use the provided UI to delegate access control for their resources to an Authorisation Manager. Such delegation had to be set up only once per user per application. A user could choose their preferred AM either with a graphical interface or by providing a URL of an AM at a simple Web input form (recall the discussion on different approaches to the problem of selecting the Authorisation Manager in Section 5.4.1). Then, a user was able to select which resources should be UMA-protected and they defined access control policies for those resources at an AM. Both applications used a custom implementation of the `HostUMADDataProvider`.

UMA/j Requester was tested similarly to UMA/j Host. In particular, this framework was added to a Web application, which was used as a Requester for the aforementioned Hosts. This application allowed a user to provide a URL of an UMA-protected resource and get access to such resource. It could access either files stored at the secure file service or photos stored at the online gallery service.

Integration of UMA/j Requester required some programming effort and changes in configuration files of the implemented requester application. Similarly to UMA/j Host, this framework was included in the classpath of the application (i.e. as a dependency in the Maven `pom.xml` file). Then, the UMA functionality was added by including the Requester Filter and Token Retrieval modules in the deployment descriptor of the application (i.e. in the `web.xml` file). Configuration of the framework was provided in an XML file.

End-users of the Requester used the implemented UI to provide a URL of the resource that they wanted to access. The Requester then tried to access such resource and referred to an AM to

Access to a protected resource

You are trying to access:

http://www.gallerify.me/a/api_v1/user/2/gallery/facebook/album/10

Smartfetch has detected that the resource you are trying to access is protected with the User-Managed Access (UMA) protocol. You will now be redirected to <http://www.smartam.net> in order to authorize **smartfetch** to access this resource on your own behalf.

Fetch authorization »

« **less**

The following information has been obtained from **Gallerify.me** about that resource that you are trying to access:

- Authorization Manager URI: <http://www.smartam.net>
- Host ID: 23663654736735673567
- Resource ID: L3VzZXlvMi9nYWxsZXJ5L2ZhY2Vib29rL2FsYnVtLzEw

Figure C.1: User Interface of a Web application accessing UMA-protected resource.

obtain authorisation (Figure C.1).

Early versions of the UMA/j Requester framework were also tested with resources that were protected with policies that required self-asserted claims to be submitted (such support was provided by SMARTAM V1). In such situations, users were presented with a custom UI, where these claims had to be confirmed. The JavaServer Faces (JSF) [35] technology was used to build the Requester's UI for self-asserted claims. Confirming these claims triggered the functionality provided by the framework's `ClaimsProcessor`. Figure 7.15 in Chapter 7 shows the User Interface of an early Requester implementation where a use could confirm claims that were later sent to the Authorisation Manager.

The framework's implementation of the `RequesterUMDataProvider` interface was used to store information regarding tokens for accessed resources in memory. It used a simple `HashMap` field to store associations between protected resources and authorisation tokens. Therefore, subsequent access requests to UMA-protected resources did not require the authorisation step, as already obtained authorisation tokens could be used. In-memory implementation of the `RequesterUMDataProvider` was sufficient for tests.

Appendix D

Puma Implementation and Evaluation

D.0.3 Implementation

Puma has been implemented in the Python programming language. In terms of programming paradigms, the library follows a functional design, as discussed in Section 6.3.2. Such design fits in well with the statelessness of the HTTP protocol [186]. Unlike the UMA/j framework, Puma provides all UMA-related functionality by itself and does not use separate libraries for discovery, dynamic registration, or OAuth 2.0.

Puma can be embedded in Python Web applications. It has been tested using applications running on the Google App Engine (GAE) platform [21]. However, the framework could be effectively used with other Python Web frameworks, such as Django [8].

Puma provides an abstraction layer, which allows applications to use various storage technologies. This layer has been introduced in order to allow Puma to be server-stack independent. Initial version of the Puma framework was tightly bound to the Google App Engine platform and its non-relational datastore [19].

D.0.4 Evaluation

Puma has been evaluated empirically from the perspective of Web application developers as well as end-users of these applications. For this purpose, two Web applications have been developed:

1. Personal Data Store (Newcastle S3P) - UMA Host;
2. CareerMonster - UMA Requester.

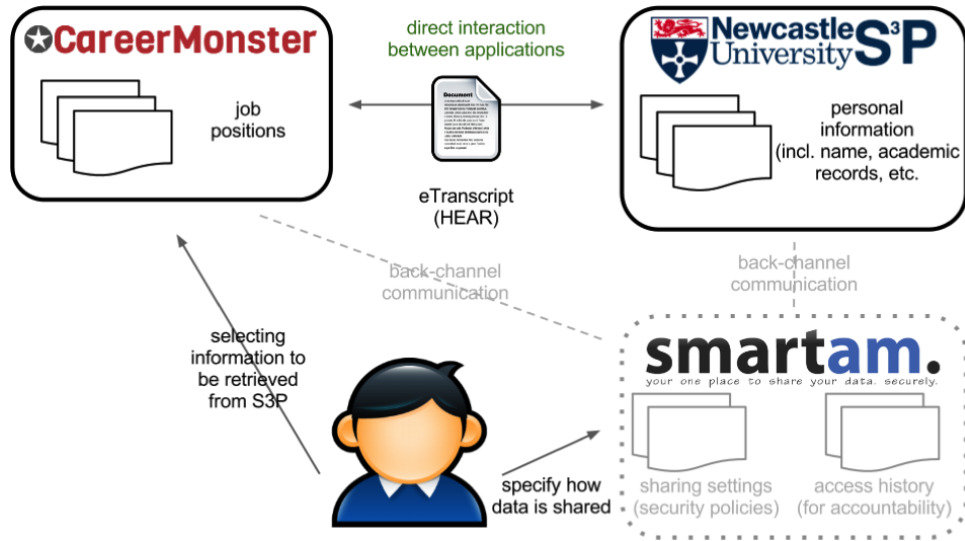


Figure D.1: Python applications integrated with SMARTAM V2 using the Puma framework.

High-level view of the implemented applications is given in Figure D.1.

D.0.4.1 Host application

A simple Personal Data Store (PDS) type application has been implemented to test the Puma framework. This application is a demo of the previously discussed Newcastle University S3P application [69], which was introduced in Section 1.1.2.

The purpose of the PDS was to provide a solution, based on the UMA proposal, to the scenario that was presented in Section 1.1.2. In particular, this solution could allow to introduce procedural and technical improvements into the existing workflow of applying for job positions by graduate students. In particular, the following improvements were used as the basis for implementing the PDS and integrating it with Puma:

1. Students should be allowed to obtain trustworthy University data easily and securely without unnecessary overheads. They should have the ability to securely share data, e.g. formatted as HEARs (Higher Education Achievement Reports) [30], directly from the University. Sharing should be done from existing systems - e.g. the aforementioned S3P, Blackboard [3], or legacy NESS (Newcastle Electronic Submission System [52]).
2. Students should be able to share trustworthy University data without the need of engaging University staff or without the need of manually handling data sharing every time such data changes. They should be allowed to establish continuous access to their educational data (HEARs, "Transcript of Records" document, etc.) without the need of contacting the school's administration staff.

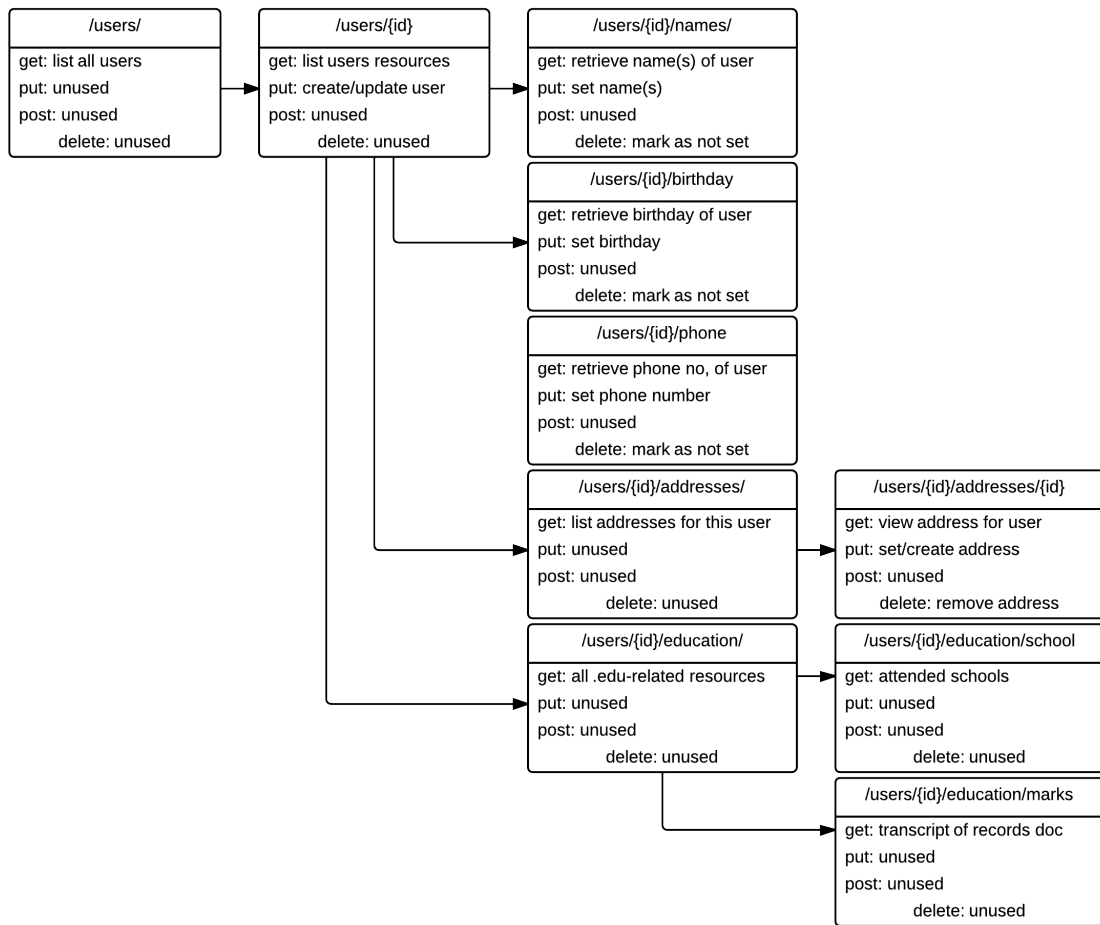


Figure D.2: UMA-protected RESTful Web API of the PDS application.

3. Third party services on the Web (employers, career services, other universities) should be able to access shared data. These services would get access to up-to-date students' academic records and personal data (students do not have to update their information during long processes of applying for graduate job positions). Access should be provided based on users' explicit consent.

The implemented PDS application allows to achieve these improvements. In particular, it allows users to store personal information, e.g. full name, date of birth, phone number, their address, attended schools and universities, as well as "Transcript of Records" documents. This application exposes data through an UMA-protected RESTful API. The user can choose the AM to protect their data and to enable selective access to this data. The API of the PDS application is visualised in Figure D.2. This application was implemented using Google's Python **webapp** framework [20]. Integration of Puma with the demo PDS application required minimal programming effort and mostly basic configuration. Firstly, this framework was included as a module in the Module

Search Path of the PDS. Secondly, a custom UI to allow users to choose their preferred AM was implemented. This UI contained a simple Web form that allowed users to provide the AM's URL. For simplicity purposes, this UI view was eventually removed and users were only concerned with authorising the PDS to use SMARTAM V2. Discovery, dynamic registration, and host introduction was performed as discussed in Sections 6.3.2.1, 6.3.2.2 and 6.3.2.3 respectively. A custom callback handler was implemented, which exchanged the authorisation code for a HAT. To allow users to register resources for protection, a resource registration and de-registration handlers provided by Puma were added to the PDS. Both handlers were included in the `url_mapping` array of the `WSGIApplication` (refer to documentation of Google App Engine in [81] for detailed information about adding handlers to Web applications). Necessary Web forms were included for both handlers. Each form provided a hidden `input` field, with the encoded resource URL that was used during resource registration. Information about successfully registered resources was stored on the PDS by Puma.

In further research, the approach described in Section 6.3.3.2.2 was used. This approach further simplified implementation of the UMA functionality at the UI level. A *"Sharing Settings"* button was displayed next to individual resources. This button directed a user to the access control policy associated with a resource and such policy was displayed within an `iframe`.

The Puma Warden component was used to protect access to resources via the API. As discussed in Section 6.3.3.3.1, the Warden is a piece of WSGI middleware suitable for use with any WSGI-compliant Python web application, and it was easily added to the application (see Listing 26). The Warden applied the UMA protocol to all access requests to protected resources (recall Listing 14). Importantly, the developer was only concerned with permission configuration (lines 6-8 in Listing 26).

```
1 routes = [ ('/api/users/(%s)/name' % r1, ApiNameHandler),
2           ('/api/users/(%s)/phone' % r1, ApiPhoneNumberHandler)
3           ... ]
4 app = webapp.WSGIApplication(routes)
5 def main():
6     permissions_map = {
7         'get': 'http://static.smartam.org/puma/read.json',
8         'post': 'http://static.smartam.org/puma/write.json' }
9     app_with_warden = Puma.Warden.Warden(app, permissions_map)
10    wsgiref.handlers.CGIHandler().run(app_with_warden)
```

Listing 26: Protecting the applications API with UMA using Puma Warden [123].

Users in the PDS were uniquely identifiable and this identifier was used for integration with

Puma entities (recall Section 6.3.3.1). Furthermore, each identifier was linked with an email address. A simple *Discovery* endpoint was implemented. This endpoint allowed a requester application to discover a user identifier based on a provided email address as a parameter of an HTTP GET request. This endpoint was not UMA protected.

D.0.4.2 Requester application

Puma Requester was evaluated by integrating it with an application named CareerMonster. CareerMonster is implemented as a demo of an online job application system. It allows students to create their personal accounts where they can apply for advertised job positions. The job application process requires submitting personal information, such as email address, full name, as well as the verified "Transcript of Records" document (recall the scenario presented in Section 1.1.2).

Additionally, the CareerMonster application allows prospective employers to create accounts and add new job positions. Importantly, employers can later view applications submitted by students and can request additional data from students. For example, employers can request access to the phone number information owned by a student and stored at the PDS.

CareerMonster can access UMA-protected resources using the Puma framework. It allows students to import their protected data from the PDS to CareerMonster. Additionally, this application can request access to UMA-protected resources, i.e. request for access can be submitted by a Requesting Party and later evaluated by an Authorising User. Such functionality is currently not a part of the UMA protocol but was added as an extra feature to Puma Requester. This feature is presented in Section 7.4.6.

Integration of Puma Requester required minimal programming effort and very basic configuration only. Similarly to Puma Host, this framework was included as a module in the Module Search Path of the Requester application. A UI was implemented to allow students to apply for advertised job positions. In particular, a simple Web form was implemented where students can provide their data either by manually filling in the form or by importing this data from their PDS. Required data included the name and the "Transcript of Records" document.

Accessing data from the PDS was done according to the UMA protocol. A custom `RequesterCallbackHandler` has been implemented. This handler was responsible for performing discovery, dynamic registration, and requester authorisation. This handler also allowed to obtain RPT tokens, which were used to access resources from the PDS. Obtaining RPT was done based on the `x-oauth_access_granted` parameter with the use of `Puma.Util.obtain_and_store_rpt_for_host_id()` method. This functionality supported the *Person-to-Self* sharing scenario (recall Section 5.3).

The *Person-to-Person* sharing scenario has been also provided using Puma. In this scenario, the employer was able to access data stored at the PDS and owned by a student. An extension to UMA has been used to allow requests for access in the absence of security policies at an AM. Implementation of this flow was provided in the `RequesterCallbackHandler` and used the `x-oauth_access_req` parameter. CareerMonster could access a protected resource only when request for this access has been approved by a student (i.e. Authorising User) at an AM.

In both aforementioned scenarios access to data did not require Requesting Parties to provide the location of resources they were trying to access (this differs from applications implemented using UMA/j). Such location was determined dynamically by the CareerMonster application using the `Discovery` endpoint available at the PDS. In particular, CareerMonster used the email address of the user who was signed in at this application to discover the identifier on the PDS. Then, the API client of the CareerMonster application used such identifier when making requests to protected resources on this PDS. This functionality was added to the previously discussed `RequesterCallbackHandler`.

Appendix E

SMARTAM V1 Data Model

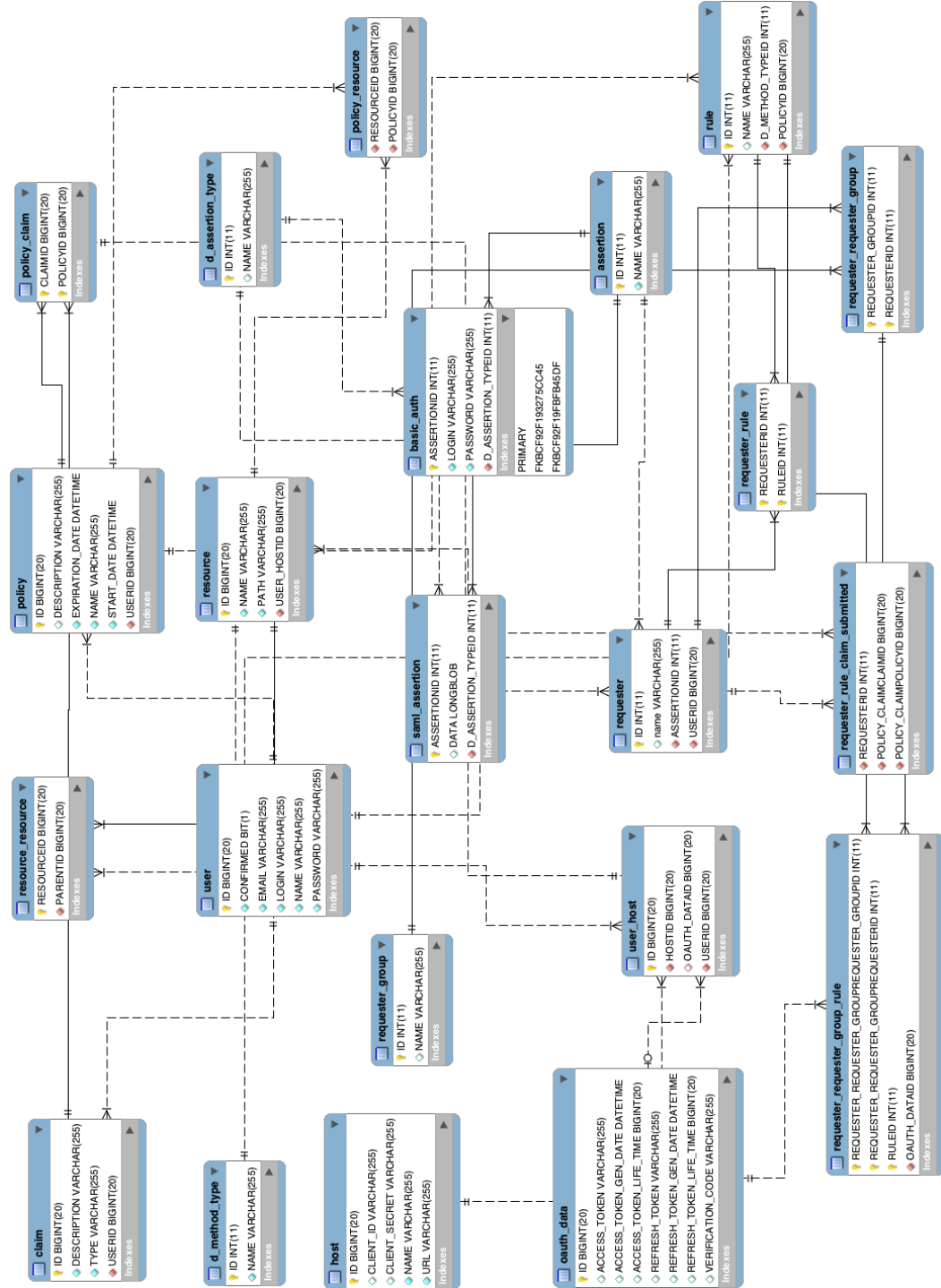


Figure E.1: SMARTAM V1 Entity Relationship Diagram.

Appendix F

SMARTAM V2 Data Model

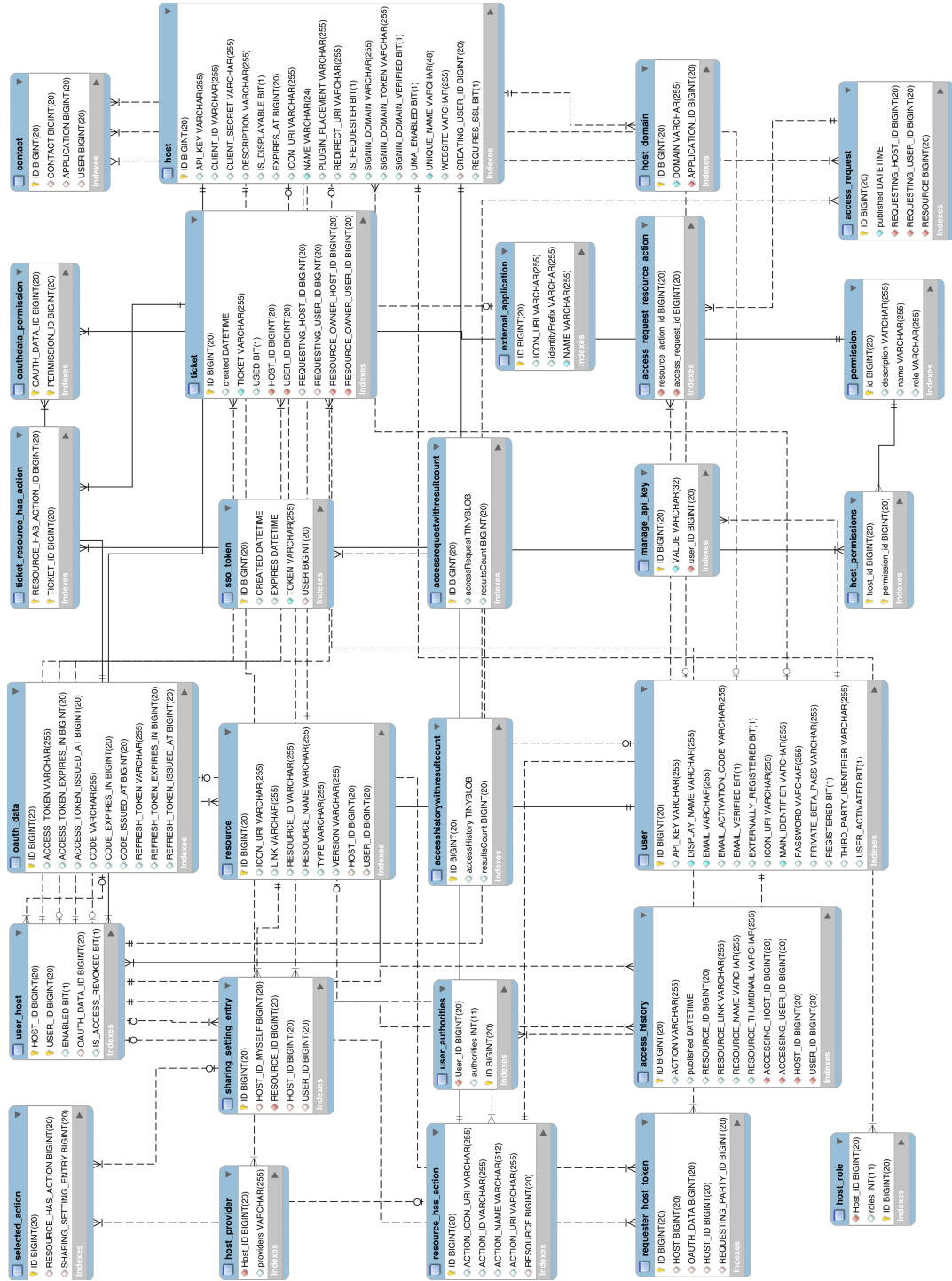


Figure F.1: SMARTAM V2 Entity Relationship Diagram.

Appendix G

Example UMA Authorisation Manager Configuration Data

```
1  { "version":"1.0", "issuer":"https://www.smartam.org",
2    "dynamic_client_registration_supported":"yes",
3    "token_types_supported":[ "artifact" ],
4    "host_grant_types_supported":[ "authorization_code",
5      "client_credentials" ],
6    "claim_types_supported":[ "openid" ],
7    "client_dynamic_registration_endpoint":
8      "https://www.smartam.org/api/oc/register",
9    "host_token_endpoint":
10     "https://www.smartam.org/oauth/token",
11    "host_user_endpoint":
12     "https://www.smartam.org/oauth/authorize",
13    "resource_set_registration_endpoint":
14     "https://www.smartam.org/api/uma/resource_reg/resource_set",
15    "token_status_endpoint":
16     "https://www.smartam.org/api/uma/validation",
17    "permission_registration_endpoint":
18     "https://www.smartam.org/api/uma/permissions_reg",
19    "requester_token_endpoint":
20     "https://www.smartam.org/oauth/token",
21    "requester_user_endpoint":
22     "https://www.smartam.org/oauth/authorize",
23    "requester_host_token_endpoint":
24     "https://www.smartam.org/api/uma/requester_host_token",
25    "permission_request_endpoint":
26     "https://www.smartam.org/api/uma/permissions_request/ticket" }
```

Listing 27: Example of Authorisation Manager Configuration Data - SMARTAM V2.

Appendix H

Questionnaire for SMARTAM V1 UI

Research Study

The questionnaire consisted of the following open and closed questions. Questions 1-7 were closed questions and could be answered in one of the following ways: *Strongly Disagree, Disagree, Neither Agree nor Disagree, Agree, Strongly Agree*. These questions were additionally accompanied by "Please Explain" subquestions. On the other hand, questions 8-9 were open while question 10 allowed to gather information about the participant.

1. The Authorisation Manager User Interface is easy to use. *Did you have any particular problems or issues with the interface while completing the task?*
2. The process of using the Authorisation Manager User Interface in order to set access control policies for requesters is clear. *Were the actions required in each step clear, or did you find the process confusing?*
3. The process of setting up access control policies has a logical order of individual steps. *Without the instructions, do you think you would have proceeded in the same way? Would you know what to do to setup a policy?*
4. I understand what it was that I was doing when using the Authorisation Manager. *Were you aware of the stage of each step when setting up access control policies?*
5. I don't notice any inconsistencies when I use the Authorisation Manager User Interface. *Did all the buttons, headings, labels, etc. react in the same consistent way? Did anything behave unexpectedly?*
6. I can recover from mistakes easily using the User Interface of Authorisation Manager.

7. There was a good use of colour in the Authorisation Manager User Interface.
8. What was your favourite feature of the Authorisation Manager User Interface? Please Explain.
9. What was your least favourite feature of the Authorisation Manager User Interface? Please Explain.
10. Please provide your: *Gender, Occupation, Age*.

Appendix I

Sharing Trustworthy Personal Data with Future Employers Scenario

This appendix shows an implemented solution for the *"Sharing Trustworthy Personal Data with Future Employers"* scenario, which was submitted to the UMA WG as well as used in the discussed research (recall Section 1.1.2). This solution comprises of two example Web applications, which were presented in Appendix D:

1. Personal Data Store, named S3P (Host);
2. Online Job Application Service, named CareerMonster (Requester).

Users can sign in to both applications using an identity provider proxy mechanism, which allows both applications to be also provisioned with the HAT/RAT tokens. Users are also automatically signed in to the Authorisation Manager. Access to data stored in S3P system is protected with our implemented SMARTAM V2. Importantly, SMARTAM will require user approval before information is shared outside of the PDS application. It will also record access requests to user's information for the purpose of accountability. SMARTAM provides a unified UI for end-users to define how data is shared, as discussed in Chapter 7.

The following steps are identified in our scenario:

1. Sean signs in to the CareerMonster website using his existing account;
2. Sean searches for the job position of his interest. He then selects the job position and applies using the provided application form;
3. Instead of manually uploading a trustworthy "Transcript of Records" document, Sean recognises that the website supports direct import from Newcastle University;

4. Sean decides to share his exam marks with CareerMonster directly from the University. He clicks on the *"Import"* button and is redirect to SMARTAM;
5. Sean authorises CareerMonster to be able to access his exam marks from S3P in a secure way. SMARTAM mediates the transaction and allows to establish a trust relationship between CareerMonster and S3P;
6. CareerMonster retrieves the "Transcript of Records" document directly from S3P, implying that this information is trustworthy and has not been forged. Sean does not have to provide a signed and stamped version of the document;
7. Sean applies for the job position by clicking the *"Submit"* button.

All steps are presented in Figure I.8 to Figure I.15.

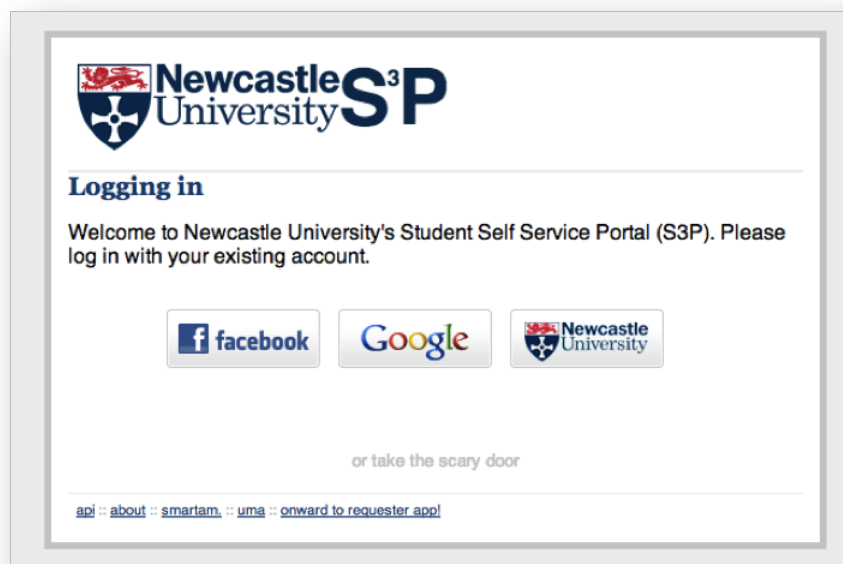


Figure I.1: Step 1 - Sean attempts to access his online data at the S3P application. Sean visits the S3P Web application at Newcastle University. Selects "Newcastle University".



Newcastle University >> Information Systems and Services >> Login Gateway

Newcastle University Login Gateway

Enter your Campus Login ID and Password

Campus Login ID:

Password:

You have been redirected to the Login Gateway by <https://smartam-login.nd.ac.uk/shibboleth/metadata>, log in here and you will be redirected to the site you were visiting.

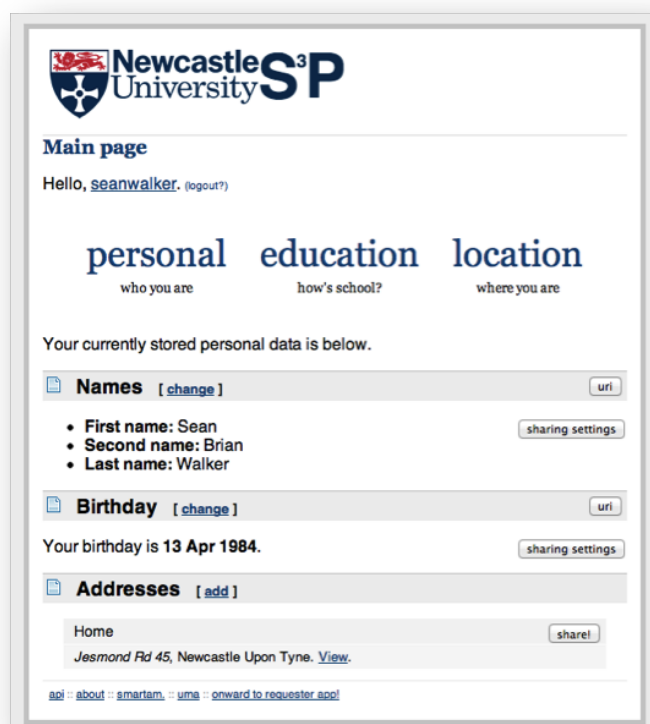
For security reasons, please exit your web browser and log out (where available) when you are finished accessing services that require authentication.

If you experience difficulty, please consult our [problems](#) page or contact it.servicedesk@nd.ac.uk

The UK Access Management Federation

Information Systems & Services
University of Newcastle upon Tyne
NE1 7RU, United Kingdom
IT Service Desk: 0191 222 5999
© 2011 Newcastle University

Figure I.2: Step 2 - Sean signs in to Newcastle University with his existing credentials.



Newcastle University S³P

Main page

Hello, [seanwalker](#). ([logout?](#))

personal **education** **location**

who you are how's school? where you are

Your currently stored personal data is below.

Names [[change](#)]

- First name: Sean
- Second name: Brian
- Last name: Walker

Birthday [[change](#)]

Your birthday is **13 Apr 1984**.

Addresses [[add](#)]

Home

Jesmond Rd 45, Newcastle Upon Tyne. [View](#)

[api](#) :: [about](#) :: [smartam](#) :: [sime](#) :: [onward to requester app!](#)

Figure I.3: Step 3 - Sean sees his personal data stored by the S3P application.

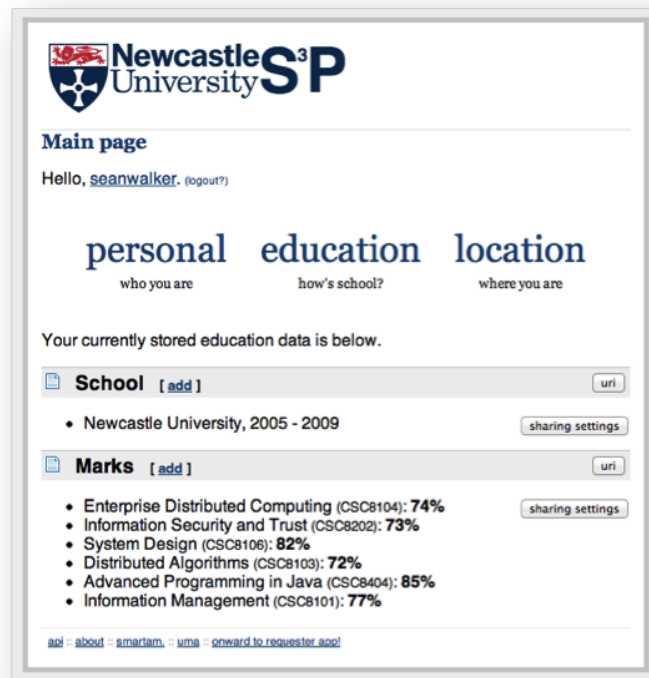


Figure I.4: Step 4 - Sean can view his academic record. Sean then attempts to apply for a job position at the CareerMonster Web application.

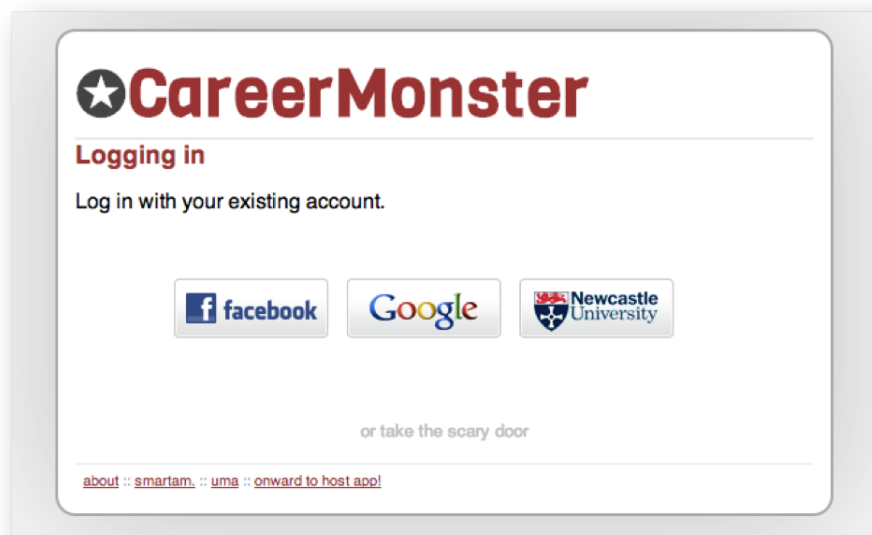


Figure I.5: Step 5 - Sean visits the CareerMonster Web application. Selects to sign in with his University's account.



Newcastle University >> Information Systems and Services >> Login Gateway

Newcastle University Login Gateway

Enter your Campus Login ID and Password

Campus Login ID:

Password:

You have been redirected to the Login Gateway by <https://smartam-login.nd.ac.uk/shibboleth/metadata>, log in here and you will be redirected to the site you were visiting.

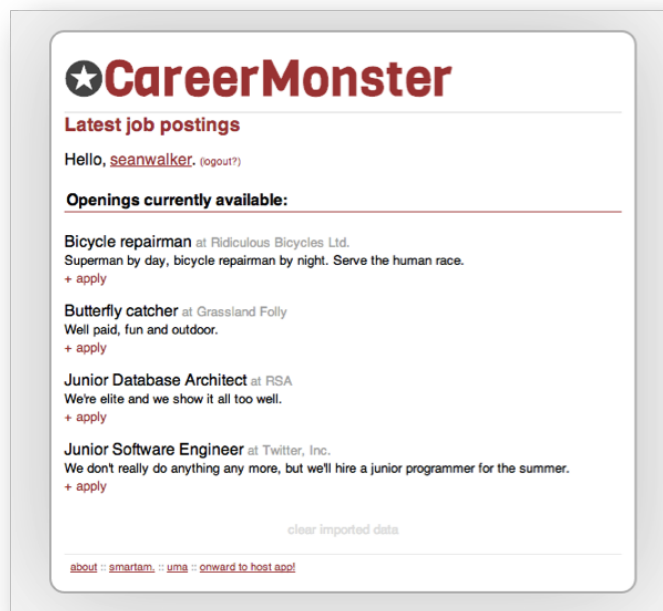
For security reasons, please exit your web browser and log out (where available) when you are finished accessing services that require authentication.

If you experience difficulty, please consult our [problems](#) page or contact it.servicedesk@nd.ac.uk

The UK Access Management Federation

Information Systems & Services
University of Newcastle upon Tyne
NE1 7RU, United Kingdom
IT Service Desk: 0191) 222 5999
© 2011 Newcastle University

Figure I.6: Step 6 - Sean signs in to Newcastle University with his existing credentials.



CareerMonster

Latest job postings

Hello, [seanwalker](#). ([logout?](#))

Openings currently available:

- Bicycle repairman** at Ridiculous Bicycles Ltd.
Superman by day, bicycle repairman by night. Serve the human race.
[+ apply](#)
- Butterfly catcher** at Grassland Folly
Well paid, fun and outdoor.
[+ apply](#)
- Junior Database Architect** at RSA
We're elite and we show it all too well.
[+ apply](#)
- Junior Software Engineer** at Twitter, Inc.
We don't really do anything any more, but we'll hire a junior programmer for the summer.
[+ apply](#)

[clear imported data](#)

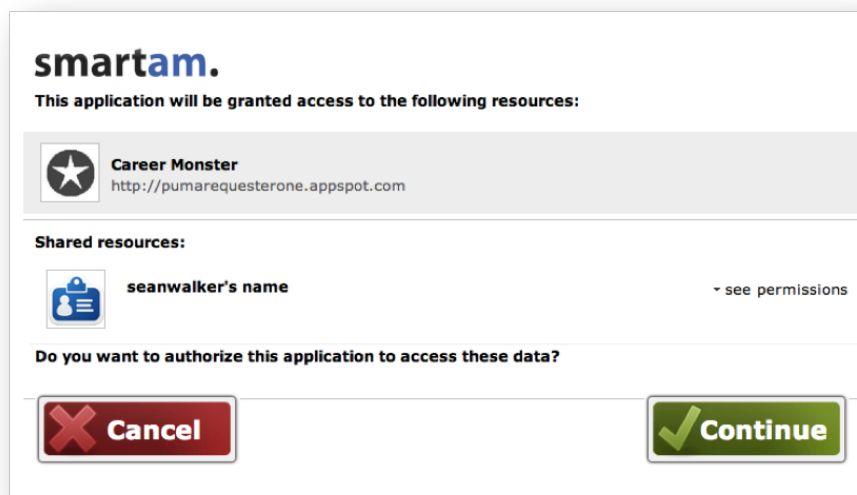
[about](#) :: [smartam](#) :: [uma](#) :: [onward to host app!](#)

Figure I.7: Step 7 - Sean is presented with a list of available job positions. He attempts to apply for the “Junior Software Engineer” position.



The image shows a web form titled "Applying for Junior Software Engineer" from CareerMonster. The form is a modal window with a red header. It contains two main sections: "Full name" and "Grades". The "Full name" section is marked as "Incomplete" and asks the user to provide their first, second, and last name, with input fields for each. Below this is a link to "import from S3P". The "Grades" section is also marked as "Incomplete" and asks the user to "Upload your Transcript of Records". It includes a "Choose File" button, a "No file chosen" status, and an "Upload" button. At the bottom of the form are buttons for "Apply for this position" and "Close".

Figure I.8: Step 8 - In order to apply for a job position, Sean has to fill in some information or import it from S3P.



The image shows a SMARTAM consent dialog. At the top is the SMARTAM logo. Below it, the text reads: "This application will be granted access to the following resources:". A box contains the Career Monster logo and the URL "http://pumarequesterone.appspot.com". Underneath, it says "Shared resources:" followed by a box containing a person icon and the text "seanwalker's name", with a link to "see permissions". The dialog asks "Do you want to authorize this application to access these data?" and has two large buttons at the bottom: a red "Cancel" button and a green "Continue" button.

Figure I.9: Step 9 - Sean is presented with a consent page, where he is required to authorize CareerMonster to access his personal data from S3P. This consent is represented on SMARTAM in form of sharing settings for Sean's name.



Figure I.10: Step 10 - Sean's personal data is retrieved by CareerMonster directly from S3P. Sean then selects to import his academic record from S3P as well.

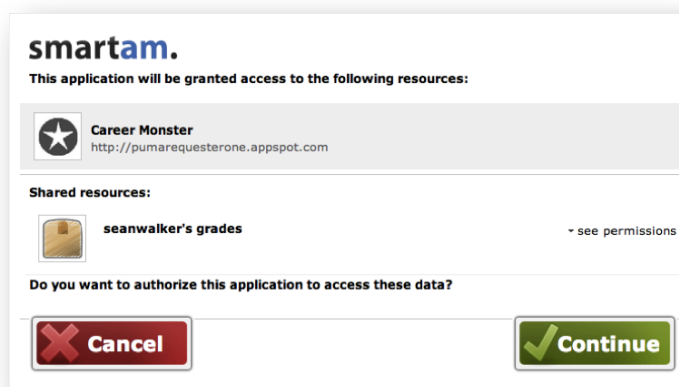


Figure I.11: Step 11 - Sean is presented with a consent page, where he is required to authorize CareerMonster to access his personal data from S3P. This consent is represented on SMART in form of additional sharing settings (this time for Sean's academic record).

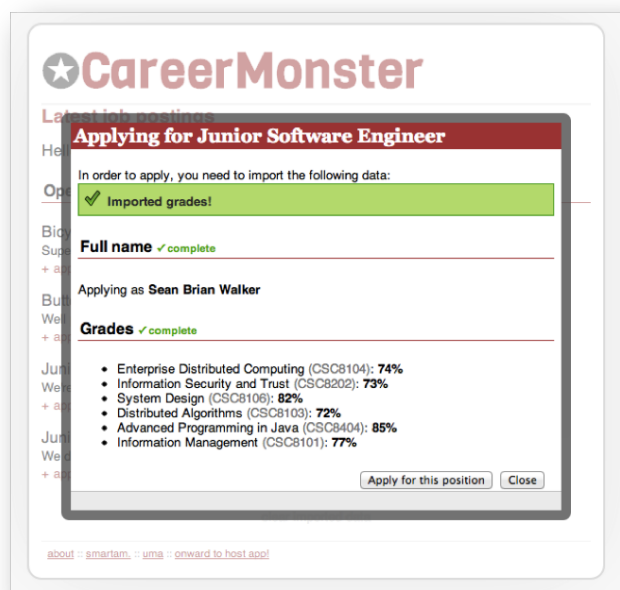


Figure I.12: Step 12 - Sean's academic record is retrieved by CareerMonster directly from S3P. At this point, all required information has been submitted to CareerMonster and Sean can apply for the job position of his choice.

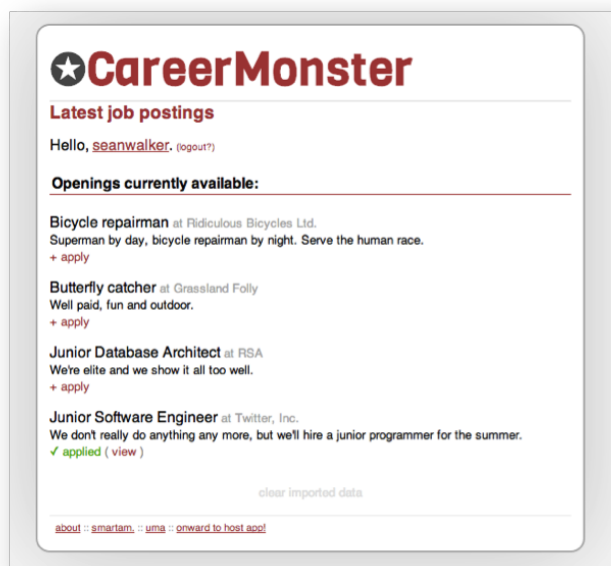


Figure I.13: Step 13 - Sean can sign in to SMART and view various information regarding his University's data and how this data is shared with other applications.

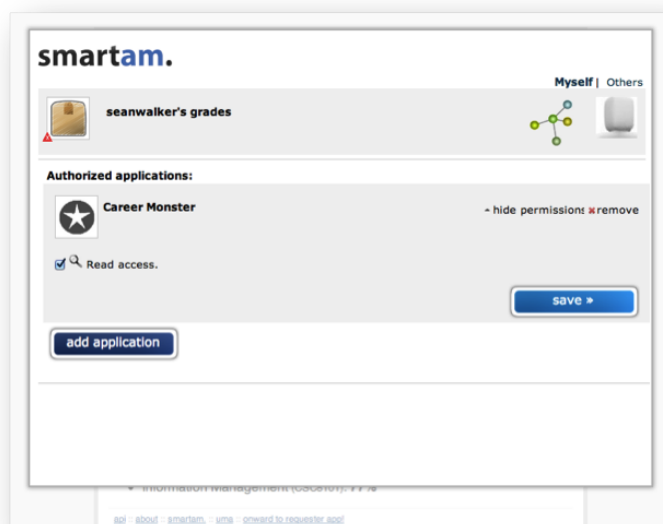


Figure I.14: Step 14 - Sean can view sharing settings for his academic record.

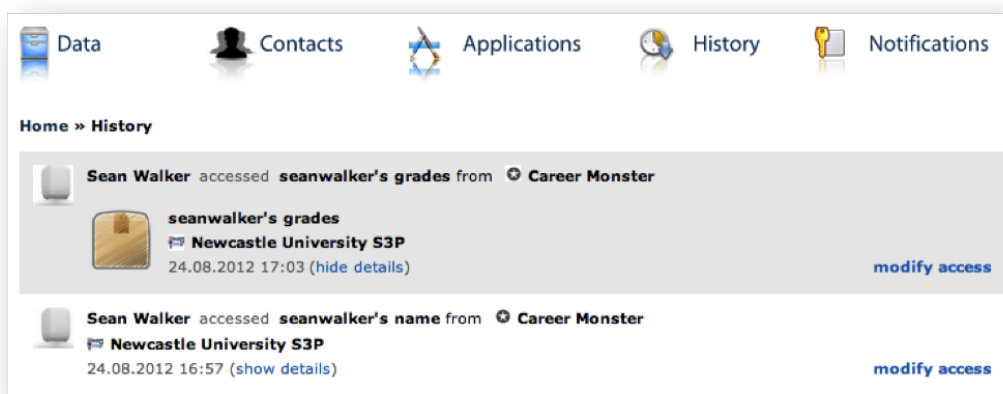


Figure I.15: Step 15 - Sean can view how his data is shared with other applications.

Appendix J

Mind map of access control for the open Web

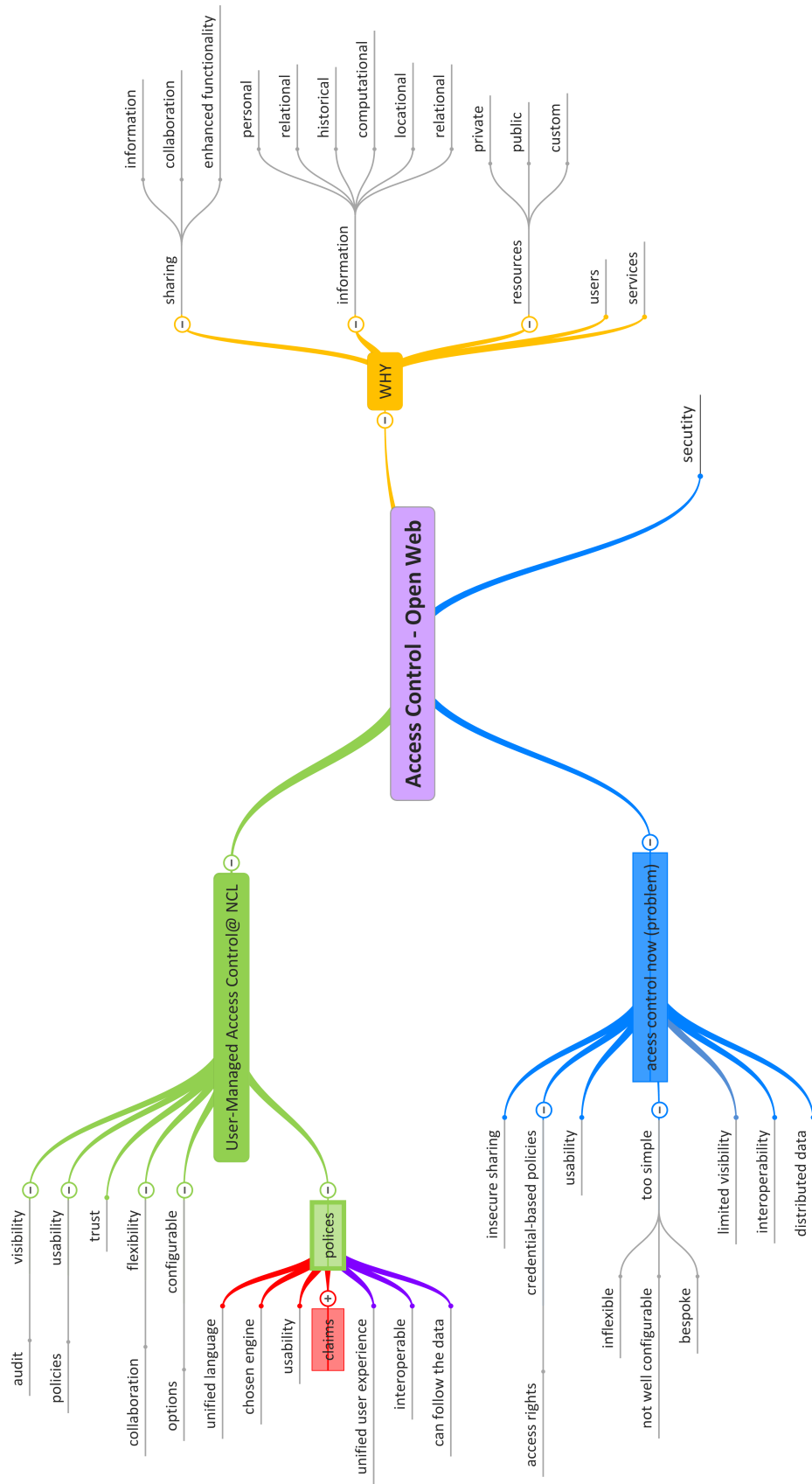


Figure J.1: Mind map of access control on the open Web - part 1.

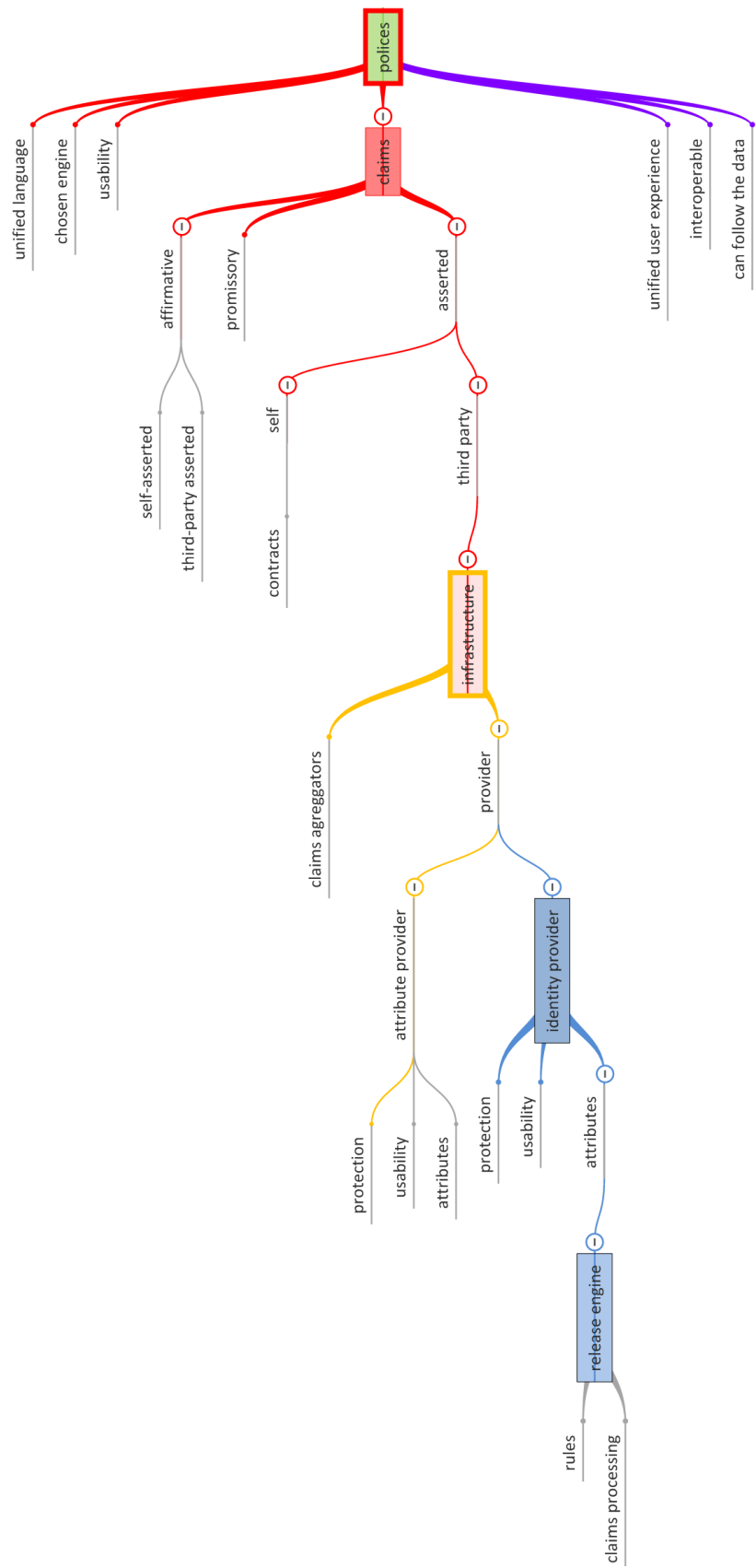


Figure J.2: Mind map of access control on the open Web - part 2.

Bibliography

- [1] 3scale: Free and Enterprise API Management Platform and Infrastructure. <http://www.3scale.com>. Accessed 16/05/2013.
- [2] Apigee: Free API Management - API analytics and control. <http://www.apigee.com>. Accessed 16/05/2013.
- [3] Blackboard. <http://blackboard.ncl.ac.uk>. Accessed 16/05/2013.
- [4] Claim. <http://wiki.idcommons.net/Claim>. Accessed 16/05/2013.
- [5] Concur Online Business Travel and Expense Management. <https://www.concur.com>. Accessed 16/05/2013.
- [6] Coursera. <https://www.coursera.org/>. Accessed 16/05/2013.
- [7] Cross-site scripting. https://en.wikipedia.org/wiki/Cross-site_scripting. Accessed 16/05/2013.
- [8] Django. Python Web Framework. <http://www.djangoproject.com>. Accessed 16/05/2013.
- [9] Facebook. <http://facebook.com>. Accessed 16/05/2013.
- [10] Facebook graph api. <https://developers.facebook.com/docs/reference/api/>. Accessed 16/05/2013.
- [11] Facebook login. <https://developers.facebook.com/docs/concepts/login/>. Accessed 16/05/2013.
- [12] FeedHenry - Mobile Application Platform. <http://www.feedhenry.com/>. Accessed 16/05/2013.
- [13] Flickr Authentication API. <http://www.flickr.com/services/api/auth.howto.web.html>. Accessed 16/05/2013.
- [14] Gigya social login. <http://www.gigya.com/>. Accessed 16/05/2013.
- [15] Gmail. <https://mail.google.com>. Accessed 16/05/2013.
- [16] Google+. <https://plus.google.com>. Accessed 16/05/2013.
- [17] Google Accounts Authentication and Authorization: AuthSub for Web Applications. <https://developers.google.com/accounts/docs/AuthSub>. Accessed 16/05/2013.
- [18] Google analytics. www.google.co.uk/analytics/. Accessed 16/05/2013.
- [19] Google App Engine - Storing Data. <https://developers.google.com/appengine/docs/python/datastore/>. Accessed 16/05/2013.

BIBLIOGRAPHY

- [20] Google App Engine - The webapp Framework. <https://developers.google.com/appengine/docs/python/tools/webapp/>. Accessed 16/05/2013.
- [21] Google AppEngine. <http://appengine.google.com>. Accessed 16/05/2013.
- [22] Google Apps for Business. <http://www.google.com/enterprise/apps/business/>. Accessed 16/05/2013.
- [23] Google Calendar. <https://calendar.google.com>. Accessed 16/05/2013.
- [24] Google Docs - Online Documents, Spreadsheets, Presentations. <https://docs.google.com>. Accessed 16/05/2013.
- [25] Google Drive. <https://drive.google.com/>. Accessed 16/05/2013.
- [26] Google Identity Toolkit. <https://developers.google.com/identity-toolkit/>. Accessed 16/05/2013.
- [27] Google+ Sign In. <https://developers.google.com/+web/signin/>. Accessed 16/05/2013.
- [28] Google street identity. <https://sites.google.com/site/streetidentitylmnop/>. Accessed 16/05/2013.
- [29] Google Street Identity. <https://sites.google.com/site/streetidentitylmnop/>. Accessed 16/05/2013.
- [30] HEAR: Higher Education Achievement Report. <http://www.hear.ac.uk/>. Accessed 16/05/2013.
- [31] Infinispan: Transactional in-memory key/value NoSQL datastore and Data Grid. <http://www.jboss.org/infinispan/>. Accessed 15/05/2013.
- [32] Janrain Engage. <http://janrain.com/products/engage/>. Accessed 16/05/2013.
- [33] Java Database Connectivity. http://en.wikipedia.org/wiki/Java_Database_Connectivity. Accessed 16/05/2013.
- [34] Java Platform, Enterprise Edition (Java EE) Technical Documentation. <http://docs.oracle.com/javaee>. Accessed 16/05/2013.
- [35] JavaServer Faces Technology. <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>. Accessed 16/05/2013.
- [36] jQuery JavaScript Library. <http://jquery.com/>. Accessed 16/05/2013.
- [37] JSON Web Encryption (JWE). <http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption>. (Work in Progress) Accessed 16/05/2013.
- [38] JSON Web Signature (JWS). <http://tools.ietf.org/html/draft-ietf-jose-json-web-signature>. (Work in Progress) Accessed 16/05/2013.
- [39] JSON Web Token (JWT). <http://tools.ietf.org/html/draft-ietf-oauth-json-web-token>. (Work in Progress) Accessed 16/05/2013.
- [40] Kantara Initiative. <http://kantarainitiative.org/>. Accessed 29/06/2010.
- [41] Kerberos Consortium. <http://www.kerberos.org>. Accessed 16/05/2013.
- [42] Liberty alliance. <http://www.projectliberty.org/>. Accessed 16/05/2013.

BIBLIOGRAPHY

- [43] Liberty ID-WSF Authentication, Single Sign-On, and Identity Mapping Services Specification. <http://www.projectliberty.org/liberty/content/download/3440/22946/file/liberty-idwsf-authn-svc-v2.0-original.pdf>. Accessed 16/05/2013.
- [44] Liferay Enterprise open source portal and collaboration software. <http://www.liferay.com>. Accessed 16/05/2013.
- [45] Locker: Service development status. <https://github.com/LockerProject/Locker/wiki/Service-development-status>. Accessed 16/05/2013.
- [46] Mashery API Management. <http://www.mashery.com>. Accessed 16/05/2013.
- [47] Maven - Apache build manager for Java project. <http://maven.apache.org>. Accessed 16/05/2013.
- [48] Maven Graph Plugin. <http://mvnplugins.fusesource.org/maven/1.0/maven-graph-plugin/index.html>. Accessed 16/05/2013.
- [49] MOOC List: Massive Open Online Courses List. <http://www.mooc-list.com/>. Accessed 16/05/2013.
- [50] [MS-ADTS]: Active Directory Technical Specification. <http://msdn.microsoft.com/en-us/library/cc223122.aspx>. Accessed 16/05/2013.
- [51] Mydex Personal Data Store. <http://mydex.org/>. Accessed 16/05/2013.
- [52] NESS: Newcastle Electronic Submission System. <http://ness.ncl.ac.uk>. Accessed 16/05/2013.
- [53] Nov 2011 Google and ID/DataWeb demo. <https://sites.google.com/site/streetidentitylmnop/workinggroup/demo2>. Accessed 16/05/2013.
- [54] OASIS Security Services (SAML) TC. <https://www.oasis-open.org/committees/security/>. Accessed 16/05/2013.
- [55] OAuth 2.0 Message Authentication Code (MAC) Tokens. <http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac>. Accessed 16/05/2013.
- [56] OpenID Authentication 1.0. <http://openid.net/specs/specs-1.0.bml>.
- [57] OpenID Connect Messages 1.0. http://openid.net/specs/openid-connect-messages-1_0.html. Accessed 16/05/2013.
- [58] OpenID Connect Specifications. <http://openid.net/connect/>. Accessed 16/05/2013.
- [59] OpenID Phishing Brainstorm. http://wiki.openid.net/w/page/12995216/OpenID_Phishing_Brainstorm. Accessed 16/05/2013.
- [60] OWASP - Open Web Application Security Project. <http://www.owasp.org>.
- [61] OWASP Enterprise Security API. http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API. Accessed 29/06/2010.
- [62] PEP 333 - Python Web Server Gateway Interface v1.0. <http://www.python.org/dev/peps/pep-0333/>.
- [63] Personal Data Ecosystem Consortium. <http://pde.cc/>. Accessed 16/05/2013.
- [64] Personal.com. <http://personal.com>. Accessed 16/05/2013.

BIBLIOGRAPHY

- [65] Pinterest. <http://pinterest.com>. Accessed 16/05/2013.
- [66] ProgrammableWeb: Web Services Directory. <http://www.programmableweb.com/apis/directory>. Accessed 16/05/2013.
- [67] Pure Templating Engine. <http://beebole.com/pure/>. Accessed 16/05/2013.
- [68] Salesforce CRM. <http://www.salesforce.com>. Accessed 16/05/2013.
- [69] Self-Service Student Portal. Newcastle University. <https://s3portal.ncl.ac.uk>. Accessed 16/05/2013.
- [70] September Google initial demo. <https://sites.google.com/site/streetidentitylmnop/workinggroup/demo1>. Accessed 16/05/2013.
- [71] Sharing Trustworthy Personal Data with Future Employers. http://kantarainitiative.org/confluence/display/uma/cv_sharing_scenario. Accessed 16/05/2013.
- [72] Shibboleth Web Single Sign-On. <http://shibboleth.internet2.edu/>. Accessed 29/06/2010.
- [73] Singly. <http://singly.com/>. Accessed 16/05/2013.
- [74] Spring Framework. <http://www.springsource.org/spring-framework>. Accessed 16/05/2013.
- [75] Spring Security. <http://www.springsource.org/spring-security>. Accessed 16/05/2013.
- [76] Spring Web MVC Framework. <http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/mvc.html>. Accessed 16/05/2013.
- [77] Student-Managed Access to Online Resources (SMART). http://www.jisc.ac.uk/whatwedo/programmes/di_directions/accessandidentitymanagement/smart.aspx. Accessed 16/05/2013.
- [78] The ID-WSF Evolution Work Group. <http://kantarainitiative.org/confluence/display/idwsf>. Accessed 29/06/2010.
- [79] The Internet Engineering Task Force (IETF). <http://www.ietf.org>. Accessed 16/05/2013.
- [80] The Locker Project. <http://lockerproject.org/>. Accessed 16/05/2013.
- [81] The WSGIApplication Class. <https://developers.google.com/appengine/docs/python/tools/webapp/wsgiapplicationclass>. Accessed 16/05/2013.
- [82] Udacity. <https://www.udacity.com/>. Accessed 16/05/2013.
- [83] UK Federation. <http://www.ukfederation.org.uk/>. Accessed 16/05/2013.
- [84] UMA Requirements. <http://kantarainitiative.org/confluence/display/uma/UMA+Requirements>. Accessed 29/06/2010.
- [85] UMA Scenarios and Use Cases. <http://kantarainitiative.org/confluence/display/uma/UMA+Scenarios+and+Use+Cases>. Accessed 29/06/2010.
- [86] User-Managed Access Work Group. <http://kantarainitiative.org/confluence/display/uma>. Accessed 16/08/2012.

BIBLIOGRAPHY

- [87] User-Managed Access Work Group Participants. <http://kantarainitiative.org/confluence/display/uma/Participant+Roster>. Accessed 16/08/2012.
- [88] Web 2.0. http://en.wikipedia.org/wiki/Web_2.0. Accessed 16/05/2013.
- [89] Web Server Gateway Interface. <http://www.wsgi.org>. Accessed 16/05/2013.
- [90] Wordpress blogging platform. <http://www.wordpress.org>. Accessed 16/05/2013.
- [91] Yahoo! Browser-Based Authentication. <http://developer.yahoo.com/bbauth/>. Accessed 16/05/2013.
- [92] YouTube. <https://www.youtube.com/>. Accessed 16/05/2013.
- [93] XML Encryption Syntax and Processing. <http://www.w3.org/TR/xmlenc-core/>, December 2002.
- [94] Web Services Architecture. <http://www.w3.org/TR/ws-arch/>, February 2004.
- [95] BS ISO/IEC 27001:2005 - Information technology - Security techniques - Information security management systems - Requirements. British Standards Institution, 2005.
- [96] BS ISO/IEC 27002:2005 - Information technology - Security techniques - Code of practice for information security management. British Standards Institution, 2005.
- [97] OASIS eXtensible Access Control Markup Language (XACML). Version 2.0. <http://www.oasis-open.org/committees/xacml/>, 2005.
- [98] SAML 2.0 profile of XACML v2.0. <http://www.oasis-open.org/committees/xacml/>, February 2005.
- [99] OASIS Reference Model for Service Oriented Architecture. Version 1.0. <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf>, October 2006.
- [100] OpenID Authentication 1.1. http://openid.net/specs/openid-authentication-1_1.html, May 2006.
- [101] JSR-000154 Java™ Servlet 2.5 Specification (Maintenance Release). <http://www.jcp.org/aboutJava/communityprocess/mrel/jsr154/>, September 2007.
- [102] OASIS Security Assertion Markup Language (SAML). Version 2.0. <http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf>, 2007.
- [103] OpenID Attribute Exchange. http://openid.net/specs/openid-attribute-exchange-1_0.html, Dec 2007.
- [104] OpenID Authentication 2.0. http://openid.net/specs/openid-authentication-2_0.txt, Dec 2007.
- [105] OpenID Authentication 2.0. http://openid.net/specs/openid-authentication-2_0.html, December 2007.
- [106] Security Policy Assertion Language SecPAL. Version 2.0. <http://research.microsoft.com/projects/SecPAL/>, 2007.
- [107] Web Services Profile of XACML (WS-XACML). Version 1.0. <http://www.oasis-open.org/committees/xacml/>, 2007.

BIBLIOGRAPHY

- [108] XACML 2.0 Interop Scenarios Working Draft. Version 0.12. <http://www.oasis-open.org/committees/download.php/24475/xacml-2.0-core-interop-draft-12-04.doc>, June 2007.
- [109] Cross-Enterprise Security and Privacy Authorization (XSPA) Profile of XACML v2.0 for Healthcare. Committee Draft. <http://www.oasis-open.org/committees/xacml/>, 2008.
- [110] Extensible Markup Language (XML) 1.0. <http://www.w3.org/TR/xml/>, November 2008.
- [111] Identity Metasystem Interoperability Version 1.0. <http://www.oasis-open.org/committees/download.php/29979/identity-1.0-spec-cd-01.pdf>, Nov 2008. Committee Draft 01.
- [112] XACML 2.0 RSA 2008 Interop Scenarios Working Draft. Version 0.12. <http://www.oasis-open.org/committees/download.php/28030/XACML-20-RSA-Interop-Documents-V-01.zip>, April 2008.
- [113] XACML v3.0 Administration and Delegation Profile. Version 1.0. <http://www.oasis-open.org/committees/xacml/>, 2008.
- [114] XML Signature Syntax and Processing. <http://www.w3.org/TR/xmlsig-core/>, June 2008.
- [115] Simple Web Token. <http://oauth-wrap-wg.googlegroups.com/web/SWT-v0.9.5.1.pdf>, Nov 2009. Version 0.9.5.1.
- [116] Extensible Resource Descriptor (XRD) Version 1.0. <http://www.oasis-open.org/committees/download.php/36473/xrd-1.0-wd14.html>, Feb 2010. Working Draft 14.
- [117] Leeloo - OAuth 2.0 Framework. <https://bitbucket.org/smartproject/oauth-2.0>, August 2010.
- [118] OAuth 2.0 support for the Kerberos V5 Authentication Protocol. <http://tools.ietf.org/html/draft-hardjono-oauth-kerberos-00>, Jun 2010. (Work in Progress).
- [119] OAuth WRAP - Web Resource Authorization Profiles. <http://tools.ietf.org/html/draft-hardt-oauth-01>, Jan 2010. (Work in Progress).
- [120] Apache Amber - OAuth 2.0 Framework. <http://incubator.apache.org/amber/>, 2011.
- [121] Kantara Initiative Announces Winners of the 2011 IDDY Awards. <http://kantarainitiative.org/kantara-initiative-announces-winners-of-the-2011-iddy-award>, February 2011.
- [122] The OAuth 1.0 Guide. <http://hueniverse.com/oauth/guide/>, July 2011.
- [123] Puma: building a host application. <http://smartjisc.wordpress.com/2012/04/20/puma-building-a-host-application/>, April 2012. Accessed 16/05/2013.
- [124] Puma: building a requester application. <http://smartjisc.wordpress.com/2012/06/30/puma-building-a-requester-application/>, June 2012. Accessed 16/05/2013.
- [125] Request / Response Interface based on JSON and HTTP for XACML 3.0 Version 1.0 - Working Draft 09. <https://www.oasis-open.org/committees/download.php/47775/xacml-json-http-v1.0-wd09.doc>, December 2012.
- [126] Street Identity Code Repository. <https://code.google.com/p/streetidentity>, March 2012.

- [127] Welcome to the API Economy. <http://www.forbes.com/sites/ciocentral/2012/08/29/welcome-to-the-api-economy/>, August 2012.
- [128] Adobe Developed Connection - Flash Developer Centre. <http://www.adobe.com/devnet/flash.html>, April 2013. Accessed 16/05/2013.
- [129] Apache Oltu - OAuth 2.0 Framework. <http://oltu.apache.org>, 2013.
- [130] Blogger. <https://www.blogger.com/>, April 2013.
- [131] Google's Internet Identity Research. <https://sites.google.com/site/oauthgoog>, May 2013. Accessed 16/05/2013.
- [132] MySQL Relational Database. <http://www.mysql.com/>, April 2013. Accessed 16/05/2013.
- [133] NIST Special Publication 800-162: Guide to Attribute Based Access Control (ABAC) Definition and Considerations (Draft). http://csrc.nist.gov/publications/drafts/800-162/sp800_162_draft.pdf, May 2013.
- [134] C. Adams and S. Lloyd. *Understanding public-key infrastructure: concepts, standards, and deployment considerations*. New Riders Pub, 1999.
- [135] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Frohner, A. Gianoli, K. Lörentey, and F. Spataro. VOMS, an Authorization System for Virtual Organizations. In F. Fernández Rivera, M. Bubak, A. Gómez Tato, and R. Doallo, editors, *Grid Computing*, volume 2970 of *Lecture Notes in Computer Science*, chapter 5, pages 33–40. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004.
- [136] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, A. Frohner, K. Lörentey, and F. Spataro. From gridmap-file to voms: Managing authorization in a grid environment. *Future Gener. Comput. Syst.*, 21(4):549–558, Apr. 2005.
- [137] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2003.
- [138] Apache Software Foundation. Apache CXF: An Open-Source Services Framework. <http://cxf.apache.org>, April 2013. Accessed 16/05/2013.
- [139] Apache Software Foundation. Apache Tomcat. <http://tomcat.apache.org>, April 2013. Accessed 16/05/2013.
- [140] C. A. Ardagna, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. A web service architecture for enforcing access control policies. *Electron. Notes Theor. Comput. Sci.*, 142:47–62, Jan. 2006.
- [141] D. Balfanz. Usable access control for the world wide web. In *ACSAC '03: Proc. of the 19th Annual Computer Security Applications Conference*, page 406, Washington, DC, USA, 2003.
- [142] T. Barton, J. Basney, T. Freeman, T. Scavo, F. Siebenlist, V. Welch, R. Ananthakrishnan, B. Baker, M. Goode, and K. Keahey. Identity Federation and Attribute-based Authorization through the Globus Toolkit, Shibboleth, Gridshib, and MyProxy. In *5th Annual PKI R&D Workshop*, Apr. 2006.
- [143] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the web. In *Proc. of the 11th USENIX Security Symposium*, pages 93–108, Berkeley, CA, USA, 2002.

- [144] M. Benantar. *Access Control Systems: Security, Identity Management and Trust Models*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [145] T. Berners-Lee. Information Management: A Proposal. Technical report, CERN, Mar. 1989.
- [146] V. Berstis. Security and protection of data in the IBM System/38. In *Proceedings of the 7th annual symposium on Computer Architecture*, ISCA '80, pages 245–252, New York, NY, USA, 1980. ACM.
- [147] E. Bertino, S. Castano, and E. Ferrari. On specifying security policies for web documents with an XML-based language. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, SACMAT '01, pages 57–65, New York, NY, USA, 2001. ACM.
- [148] R. Bhatti, A. Ghafoor, E. Bertino, and J. B. D. Joshi. X-GTRBAC: an XML-based policy specification framework and architecture for enterprise-wide access control. *ACM Trans. Inf. Syst. Secur.*, 8(2):187–227, May 2005.
- [149] R. Botha and J. H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3):666–682, 2001.
- [150] D. Brewer and M. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206 –214, may 1989.
- [151] S. Cantor et al. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, Mar. 2005.
- [152] B. Carminati, E. Ferrari, and A. Perego. Rule-based access control for social networks. In *Proceedings of the 2006 international conference on On the Move to Meaningful Internet Systems: AWeSOMe, CAMS, COMINF, IS, KSinBIT, MIOS-CIAO, MONET - Volume Part II*, OTM'06, pages 1734–1744, Berlin, Heidelberg, 2006. Springer-Verlag.
- [153] I. Carter. A Research Study to Evaluate the Usability Of a Proposed User Managed Access Service User Interface. Master's thesis, School of Computing Science, Newcastle University, October 2010.
- [154] D. Catalano. Extending UMA Protocol to Support Trusted Claims Approach. http://www.projectliberty.org/confluence/download/attachments/17301540/Trusted+Claim+Model+in+UMA_v3.2.pdf, August 2010. Accessed 16/05/2013.
- [155] D. Catalano. Extending UMA Protocol to Support Trusted Claims (tClaims). <http://www.slideshare.net/domcat/uma-trusted-claims>, July 2011. Accessed 16/05/2013.
- [156] D. Catalano. UMA: Measuring elements of Trust. <http://kantarainitiative.org/confluence/display/uma/Measuring+elements+of+Trust>, April 2011. Accessed 20/05/2014.
- [157] D. Catalano. UMA Trust Model User guide. <http://kantarainitiative.org/confluence/display/uma/UMA+Trust+Model+User+guide>, May 2012. Accessed 16/05/2013.
- [158] A. Cavoukian. Privacy in the clouds. *Identity in the Information Society*, 1:89–108, 2008.
- [159] V. Cerf, Y. Dalal, and C. Sunshine. Specification of Internet Transmission Control Program. RFC 675, Dec. 1974.

BIBLIOGRAPHY

- [160] D. Chadwick. GFD.156 Functional Components of Grid Service Provider Authorisation Service Middleware, October 2009.
- [161] D. Chadwick and L. Su. GFD.157 Use of WS-TRUST and SAML to access a Credential Validation Service, November 2009.
- [162] D. W. Chadwick, G. Inman, and N. Klingenstein. A conceptual model for attribute aggregation. *Future Gener. Comput. Syst.*, 26(7):1043–1052, July 2010.
- [163] D. W. Chadwick and S. F. Lievens. Enforcing "sticky" security policies throughout a distributed application. In *Proceedings of the 2008 Workshop on Middleware Security, MidSec '08*, pages 1–6, New York, NY, USA, 2008. ACM.
- [164] A. Chakrabarti. *Grid Computing Security*. Springer-Verlag Berlin Heidelberg, 2007.
- [165] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Definition Language (WSDL). <http://www.w3.org/TR/wsd1>, March 2001.
- [166] D. Crockford. JavaScript Object Notation (JSON). <http://www.json.org>, 2006.
- [167] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.
- [168] C. de Laat, G. Gross, L. Gommans, J. Vollbrecht, and D. Spence. Generic AAA Architecture. RFC 2903 (Experimental), Aug. 2000.
- [169] Y. Demchenko, O. Koeroo, C. de Laat, and H. Sagehaug. Extending xacml authorisation model to support policy obligations handling in distributed application. In *Proceedings of the 6th international workshop on Middleware for grid computing, MGC '08*, pages 5:1–5:6, New York, NY, USA, 2008. ACM.
- [170] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '06*, pages 581–590, New York, NY, USA, 2006. ACM.
- [171] V. Dhankhar, S. Kaushik, and D. Wijesekera. XACML Policies for Exclusive Resource Usage. pages 275–290. 2007.
- [172] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.
- [173] F. Dillema, S. Lupetti, and T. Stabell-Kulo. A decentralized authorization architecture. In *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, volume 1, pages 497 –504, may 2007.
- [174] M. Driver, R. Valdes, and G. Phifer. Rich internet applications are the next evolution of the web. *Gartner Research*, 2005.
- [175] T. Erl. *Service-Oriented Architecture: Concepts, Technology, And Design*. Pearson Education, 2006.
- [176] Eve Maler. ProtectServe draft protocol flows. <http://www.xmlgrrl.com/blog/2009/04/02/protectserve-draft-protocol-flows/>, Mar 2009. Accessed 29/06/2010.
- [177] Eve Maler. To protect and to serve. <http://www.xmlgrrl.com/blog/2009/03/23/to-protect-and-to-serve/>, Mar 2009. Accessed 29/06/2010.
- [178] Eve Maler. UMA Lexicon. <http://kantarainitiative.org/confluence/display/uma/Lexicon>, April 2010. Accessed 16/05/2013.

- [179] Eve Maler. A new venn of access control for the API economy. http://blogs.forrester.com/eve_maler/12-03-12-a_new_venn_of_access_control_for_the_api_economy, March 2012. Accessed 16/05/2013.
- [180] Eve Maler. Distributed Authorization ...as conceived by UMA. <http://kantarainitiative.org/confluence/download/attachments/17760302/UMA+and+XACML+2012-10-18.pdf>, October 2012. Accessed 16/05/2013.
- [181] S. Farrell and R. Housley. An Internet Attribute Certificate Profile for Authorization. RFC 3281 (Draft Standard), June 2002.
- [182] S. Farrell, J. Vollbrecht, P. Calhoun, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. AAA Authorization Requirements. RFC 2906 (Informational), Aug. 2000.
- [183] M. S. Ferdous and R. Poet. Analysing attribute aggregation models in federated identity management. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 181–188, New York, NY, USA, 2013. ACM.
- [184] D. Ferraiolo, D. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House Computer Security Series. Artech House, 2003.
- [185] A. Field and G. Hole. *How to Design and Report Experiments*. SAGE Publications, 2003.
- [186] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.
- [187] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [188] J. Fontana. OAuth quickly moves into maturity cycle. <https://www.pingidentity.com/blogs/pingtalk/2013/05/oauth-quickly-spins-into-maturity-cycle-1.html>, May 2013. Accessed 16/05/2013.
- [189] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, Aug. 2001.
- [190] F. Fowler. *Survey research methods*. Applied social research methods series. Sage Publications, 2002.
- [191] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. <http://martinfowler.com/articles/injection.html>, January 2004.
- [192] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.
- [193] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), Aug. 2011.
- [194] C. Gates. Access control requirements for web 2.0 security and privacy. In *W2SP '07: Proc. of the Workshop on Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2007.
- [195] R. Geambasu et al. Organizing and sharing distributed personal web-service data. In *WWW '08: Proc. of the 17th Intl. Conference on World Wide Web*, pages 755–764, New York, NY, USA, 2008.

- [196] R. Geambasu, S. D. Gribble, and H. M. Levy. Cloudviews: communal data sharing in public clouds. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.
- [197] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.
- [198] R. Gupta. Security in a SOA. SOA World Magazine 7, 2007.
- [199] Y. Gurevich and I. Neeman. DKAL: Distributed-Knowledge Authorization Language. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, volume 0, pages 149–162, Washington, DC, USA, 2008. IEEE Computer Society.
- [200] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), Apr. 2010. Obsoleted by RFC 6749.
- [201] E. Hammer-Lahav and B. Cook. Web Host Metadata. RFC 6415 (Proposed Standard), Oct. 2011.
- [202] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), Oct. 2012.
- [203] M. Hart, R. Johnson, and A. Stent. More content - less control: Access control in the web 2.0. In *WOSP '08: Proc. of the first workshop on Online social networks*, pages 43–48, New York, NY, USA, 2008.
- [204] R. Hasan, M. Winslett, R. Conlan, B. Slesinsky, and N. Ramani. Please permit me: Stateless delegated authorization in mashups. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 173–182, 2008.
- [205] A. Herzberg and A. Jbara. Security and identification indicators for browsers against spoofing and phishing attacks. *ACM Trans. Internet Technol.*, 8(4):16:1–16:36, Oct. 2008.
- [206] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. RFC 2459 (Proposed Standard), Jan. 1999. Obsoleted by RFC 3280.
- [207] F. Hsu and H. Chen. Secure File System Services for Web 2.0 Applications. In *CCSW '09: Proc. of the 2009 ACM Workshop on Cloud Computing Security*, pages 11–18, New York, NY, USA, 2009. ACM.
- [208] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, Sept. 1989.
- [209] R. Johnson, E. Gamma, J. Vlissides, and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [210] M. Jones and D. Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750 (Proposed Standard), Oct. 2012.
- [211] J. Joshi. Access-control language for multidomain environments. *Internet Computing, IEEE*, 8(6):40 – 50, nov.-dec. 2004.
- [212] J. Joshi, A. Ghafoor, W. G. Aref, and E. H. Spafford. Digital government security infrastructure design challenges. *Computer*, 34(2):66–72, Feb. 2001.
- [213] M. B. Juric, I. Rozman, B. Brumen, M. Colnarić, and M. Hericko. Comparison of performance of web services, ws-security, rmi, and rmi-ssl. *J. Syst. Softw.*, 79(5):689–700, May 2006.

- [214] H. Kamoda, A. Hayakawa, M. Yamaoka, S. Matsuda, K. Broda, and M. Sloman. Policy conflict analysis using tableaux for on demand vpn framework. In *Proceedings of the First International IEEE WoWMoM Workshop on Trust, Security and Privacy for Ubiquitous Computing - Volume 03*, WOWMOM '05, pages 565–569, Washington, DC, USA, 2005. IEEE Computer Society.
- [215] R. Kanneganti and P. Chodavarapu. *SOA Security*. Manning Pubs Co Series. Manning Publications Company, 2008.
- [216] A. H. Karp, H. Haury, and M. H. Davis. From ABAC to ZBAC: the evolution of access control models. *Hewlett-Packard Development Company, LP*, 21, 2009.
- [217] A. H. Karp and M. Stiegler. Making policy decisions disappear into the user's workflow. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '10, pages 3247–3252, New York, NY, USA, 2010. ACM.
- [218] Y. Keleta, M. Coetzee, J. Eloff, and H. Venter. Proposing a Secure XACML architecture ensuring privacy and trust. In *ISSA: Information Security South Africa*, Sandton, South Africa, 2005.
- [219] F. Kerschbaum and P. Robinson. Security architecture for virtual organizations of business web services. *Journal of Systems Architecture*, 55(4):224–232, Apr. 2009.
- [220] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). RFC 1510 (Historic), Sept. 1993. Obsoleted by RFCs 4120, 6649.
- [221] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997. Updated by RFC 6151.
- [222] S. Krug. *Don't make me think!: a common sense approach to Web usability*. Voices That Matter Series. New Riders, 2006.
- [223] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, 1992.
- [224] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974.
- [225] B. Laurie. OpenID: Phishing Heaven. <http://www.links.org/?p=187>, January 2007. Accessed 16/05/2013.
- [226] A. Lee and M. Winslett. Towards standards-compliant trust negotiation for web services. In Y. Karabulut, J. Mitchell, P. Herrmann, and C. Jensen, editors, *Trust Management II*, volume 263 of *IFIP – The International Federation for Information Processing*, pages 311–326. Springer US, 2008.
- [227] A. J. Lee, M. Winslett, and et al. Traust: A trust negotiation-based authorization service for open systems. In *IN SACMAT '06: PROCEEDINGS OF THE ELEVENTH ACM SYMPOSIUM ON ACCESS*, pages 39–48. ACM Press, 2006.
- [228] H. Lee. Unraveling decentralized authorization for multi-domain collaborations. In *Collaborative Computing: Networking, Applications and Worksharing, 2007. CollaborateCom 2007. International Conference on*, pages 33 –40, nov. 2007.
- [229] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin. Access control policy combining: theory meets practice. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, pages 135–144, New York, NY, USA, 2009. ACM.

- [230] W. Lin. A Scope Attack against OAuth 2.0. <http://www.ietf.org/mail-archive/web/oauth/current/msg08409.html>, February 2012. Accessed 16/05/2013.
- [231] J. Linn. Practical authentication for distributed computing. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 31–40, May 1990.
- [232] M. Liu, W. Zhang, and H.-L. Liu. Specification of access control policies for web services. In *Computational Intelligence and Security Workshops, 2007. CISW 2007. International Conference on*, pages 472–475, dec. 2007.
- [233] T. Lodderstedt, M. McGloin, and P. Hunt. OAuth 2.0 Threat Model and Security Considerations. RFC 6819 (Informational), Jan. 2013.
- [234] M. Lorch, B. Cowles, R. Baker, L. Gommans, P. Madsen, A. McNab, L. Ramarkrishnan, K. Sankar, D. Skow, and M. Thompson. GFD.38 Conceptual grid authorization framework and classification, 2004.
- [235] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using xacml for access control in distributed systems. In *Proceedings of the 2003 ACM workshop on XML security, XMLSEC '03*, pages 25–37, New York, NY, USA, 2003. ACM.
- [236] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869, Nov. 1999.
- [237] M. P. Machulak. OAuth Dynamic Binding. http://kantarainitiative.org/confluence/download/attachments/17301540/dynamic_registration.pdf, July 2010. Accessed 16/05/2013.
- [238] M. P. Machulak, E. L. Maler, D. Catalano, and A. van Moorsel. User-Managed Access to Web Resources. In *Proceedings of the 6th ACM workshop on Digital identity management*, pages 35–44. ACM, 2010.
- [239] M. P. Machulak, Ł. Moreń, and A. van Moorsel. Design and Implementation of User-managed Access Framework for Web 2.0 Applications. In *Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing*, pages 1–6. ACM, 2010.
- [240] M. P. Machulak, S. Parkin, and A. van Moorsel. Architecting Dependable Access Control Systems for Multi-domain Computing Environments. *Architecting Dependable Systems VI*, pages 49–75, 2009.
- [241] M. P. Machulak and A. van Moorsel. A Novel Approach to Access Control for the Web. Technical Report CS-TR-1157, Newcastle University, July 2009.
- [242] M. P. Machulak and A. van Moorsel. A Novel Approach to Access Control for the Web. *IEEE SSP'09: Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009. Poster Abstract.
- [243] M. P. Machulak and A. van Moorsel. Use Cases for User-Centric Access Control for the Web. Technical Report CS-TR-1165, Newcastle University, August 2009.
- [244] M. P. Machulak and A. van Moorsel. Architecture and Protocol for User-Controlled Access Management in Web 2.0 Applications. *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems Workshops*, pages 62–71, 2010.
- [245] E. Maler. The ProtectServe and Relationship Manager Proposition. <http://xmlgrrl.com/publications/ProtectServe-IIW8-19May2009.pdf>, May 2009.

- [246] E. Maler and T. Hardjono. Binding Obligations on User-Managed Access (UMA) Participants. <http://tools.ietf.org/html/draft-maler-oauth-umatrust-00>, January 2013. Work in Progress.
- [247] E. L. Maler and P. C. Bryan. Claims 2.0. <http://kantarainitiative.org/confluence/display/uma/Claims+2.0>. Accessed 29/06/2010.
- [248] L. Martin. Usability analysis and visualization of web 2.0 applications. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 121–124, October 2008.
- [249] D. Mashima, D. Bauer, M. Ahamad, and D. M. Blough. User-centric identity management architecture using credential-holding identity agents. *Digital Identity and Access Management: Technologies and Frameworks*, 2011.
- [250] E. e. Michiels. ISO/IEC 10181-3:1996 Information technology Open Systems Interconnection Security frameworks for open systems: Access control framework. ISO/IEC, Geneva, Int. Standard Edition, 1996.
- [251] S. Murugesan. Understanding Web 2.0. *IT Professional*, 9(4):34–41, 2007.
- [252] A. Nadalin, M. Goodner, M. Gudgin, D. Turner, A. Barbir, and H. Granqvist. WS-Trust 1.4. OASIS Standard, Feb. 2009.
- [253] M. Naedele. Standards for XML and Web Services Security. *Computer*, 36(4):96–98, 2003.
- [254] B. Neuman. Proxy-based authorization and accounting for distributed systems. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 283–291, May 1993.
- [255] C. Neuman, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Draft Standard), 2005.
- [256] J. Nielsen and H. Loranger. *Prioritizing Web usability*. Voices Series. New Riders, 2006.
- [257] J. Nielsen and H. Loranger. Usability 101. <http://www.useit.com/alertbox/20030825.html>, 2006.
- [258] J. Nielsen and R. Mack. *Usability inspection methods*. Tutorial / Interact '95. Wiley, 1994.
- [259] J. Nielsen and K. Pernice. *Eyetracking Web Usability*. Voices That Matter. Pearson Education, 2010.
- [260] M. Nottingham and E. Hammer-Lahav. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785 (Proposed Standard), Apr. 2010.
- [261] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta. Comparison of json and xml data interchange formats: A case study. *Computer Applications in Industry and Engineering (CAINE)*, pages 157–162, 2009.
- [262] N. I. of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. <http://www.itl.nist.gov/fipspubs/fip180-1.htm>, April 1995.
- [263] OpenID Foundation. Account Chooser. <https://www.accountchooser.net/>. Accessed 16/05/2013.
- [264] OpenID Foundation. OpenID Connect Dynamic Client Registration. http://openid.net/specs/openid-connect-registration-1_0.html, May 2012.

- [265] A. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Bloomsbury Academic, 1992.
- [266] T. O'Reilly. What Is Web 2.0. Design Patterns and Business Models for the Next Generation of Software. <http://oreilly.com/web2/archive/what-is-web-20.html>, Sep 2005. Accessed 29/06/2010.
- [267] R. Paul. OAuth and OAuth WRAP: defeating the password anti-pattern. <http://arstechnica.com/open-source/guides/2010/01/oauth-and-oauth-wrap-defeating-the-password-anti-pattern.ars>, Jan 2010. Accessed 29/06/2010.
- [268] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, POLICY '02, pages 50–, Washington, DC, USA, 2002. IEEE Computer Society.
- [269] S. Pearson and M. C. Mont. Sticky policies: An approach for managing privacy across multiple parties. *Computer*, 44(9):60–68, Sept 2011.
- [270] D. J. Power, E. A. Politou, M. A. Slaymaker, and A. C. Simpson. Towards secure grid-enabled healthcare: Research articles. *Softw. Pract. Exper.*, 35(9):857–871, July 2005.
- [271] J. C. Process. JSR 311: JAX-RS: The Java™ API for RESTful Web Services. <http://jcp.org/en/jsr/detail?id=311>, November 2009. Accessed 16/05/2013.
- [272] D. Recordon and D. Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, pages 11–16. ACM, 2006.
- [273] F. Ricciardi. Kerberos protocol tutorial. <http://www.kerberos.org/software/tutorial.html>, November 2007. Accessed 16/05/2013.
- [274] J. Richer, J. Bradley, M. Jones, and M. P. Machulak. OAuth Dynamic Client Registration Protocol. <http://tools.ietf.org/html/draft-ietf-oauth-dyn-reg>, December 2012. Work in Progress.
- [275] P. Samarati and S. Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Berlin Heidelberg, 2001.
- [276] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, feb 1996.
- [277] R. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.
- [278] R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys (CSUR)*, 28(1):241–243, 1996.
- [279] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 51–65, 2007.
- [280] J. Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511 (Proposed Standard), June 2006.

- [281] A. Simpson. On the need for user-defined fine-grained access control policies for social networking applications. In *SOSOC '08: Proc. of the Workshop on Security in Opportunistic and SOCial networks*, pages 1–8, New York, NY, USA, 2008.
- [282] A. C. Simpson, D. J. Power, D. Russell, M. A. Slaymaker, G. Kouadri-Mostefaoui, X. Ma, and G. Wilson. A healthcare-driven framework for facilitating the secure sharing of data across organisational boundaries. In *Proceedings of HealthGrid 2008*, 2008.
- [283] M. Slaymaker, D. Power, D. Russell, and A. Simpson. On the facilitation of fine-grained access to distributed healthcare data. In W. Jonker and M. Petković, editors, *Secure Data Management*, volume 5159 of *Lecture Notes in Computer Science*, pages 169–184. Springer Berlin Heidelberg, 2008.
- [284] D. K. Smetters. Building Secure Mashups. In *W2SP '08: Proc. of the Workshop on Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2008.
- [285] K. Smith. SOA Access Control Policy Management. Approaches, Common Pitfalls, and Best Practices. <http://soa.sys-con.com/node/284576>, October 2006.
- [286] S.-T. Sun, K. Hawkey, and K. Beznosov. Secure web 2.0 content sharing beyond walled gardens. In *ACSAC '09: Proc. of the 25th Annual Computer Security Applications Conference*, pages 409–418, Dec 2009.
- [287] J. Sunshine, S. Egelman, H. Almuhiemedi, N. Atri, and L. F. Cranor. Crying wolf: an empirical study of ssl warning effectiveness. In *Proceedings of the 18th conference on USENIX security symposium, SSYM'09*, pages 399–416, Berkeley, CA, USA, 2009. USENIX Association.
- [288] The Stationery Office Limited. Data Protection Act 1998, 1998.
- [289] A. Tootoonchian et al. Lockr: social access control for web 2.0. In *WOSP '08: Proc. of the First Workshop on Online Social Networks*, pages 43–48, New York, NY, USA, 2008.
- [290] M. S. Torsten Lodderstedt, Stefanie Dronia. Token Revocation. <http://tools.ietf.org/html/draft-lodderstedt-oauth-revocation-04>, March 2012. (Work in Progress).
- [291] P. van Schaik and J. Ling. Design parameters of rating scales for web sites. *ACM Trans. Comput.-Hum. Interact.*, 14(1), May 2007.
- [292] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. AAA Authorization Application Examples. RFC 2905 (Informational), Aug. 2000.
- [293] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. AAA Authorization Framework. RFC 2904 (Informational), Aug. 2000.
- [294] W3C. SOAP. Technical Reports. Version 1.2. <http://www.w3.org/TR/soap/>, April 2007.
- [295] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '92*, pages 1–14, New York, NY, USA, 1992. ACM.
- [296] G. Wang. Improving data transmission in web applications via the translation between xml and json. In *Communications and Mobile Computing (CMC), 2011 Third International Conference on*, pages 182–185, 2011.

BIBLIOGRAPHY

- [297] U. WG. UMA 1.0 Core Protocol. <http://kantarainitiative.org/confluence/display/uma/UMA+1.0+Core+Protocol>. Accessed 17/08/2012 (Work in Progress).
- [298] D. Winer. XML-RPC Specification. <http://xmlrpc.scripting.com/spec.html>, June 1999.
- [299] W. Winsborough, K. Seamons, and V. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 1, pages 88 –102 vol.1, 2000.
- [300] T. Woo and S. Lam. Designing a distributed authorization service. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 419 –429 vol.2, March - April 1998.
- [301] M. Wu, R. C. Miller, and S. L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 601–610, New York, NY, USA, 2006. ACM.
- [302] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control. RFC 2753 (Informational), Jan. 2000.
- [303] Y. Zhang, S. Egelman, L. Cranor, and J. Hong. Phinding phish: Evaluating anti-phishing tools. In *In Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.